The J2EE™ 1.4 Tutorial

for NetBeans™ IDE 4.1

For Sun Java System Application Server Platform Edition 8.1

Eric Armstrong
Jennifer Ball
Stephanie Bodoff
Debbie Bode Carson
Ian Evans
Kenneth Ganfield
Dale Green
Kim Haase
Eric Jendrock
John Jullion-Ceccarelli
Geertjan Wielenga

May 11, 2005

Contents

About This Tutorial

THE J2EE™ 1.4 Tutorial is a guide to developing enterprise applications for the Java 2 Platform, Enterprise Edition (J2EE) version 1.4. Here we cover all the things you need to know to make the best use of this tutorial.

Who Should Use This Tutorial

This tutorial is intended for programmers who are interested in developing and deploying J2EE 1.4 applications on the Sun Java System Application Server Platform Edition 8.1 2005Q1.

Prerequisites

Before proceeding with this tutorial you should have a good knowledge of the Java programming language. A good way to get to that point is to work through all the basic and some of the specialized trails in *The Java™ Tutorial*, Mary Campione et al., (Addison-Wesley, 2000).

About the Examples

This section tells you everything you need to know to install, build, and run the examples.

Required Software

Tutorial Bundle

The tutorial example source is contained in the tutorial bundle. If you are viewing this online, you need to download tutorial bundle from:

http://www.netbeans.org/files/documents/4/441/j2eetutorial14.zip

After you have installed the tutorial bundle, the example source code is in the <*INSTALL*>/j2eetutorial14/examples/ directory.

NetBeans IDE 4.1

The tutorial examples are developed and built using the 4.1 release of the NetBeans Integrated Development Environment (IDE). The IDE is the first open source IDE to support the new J2SE 5.0 "Tiger" language features, and is the first IDE to base its project system entirely on Apache Ant. This robust open source Java IDE has everything that you need to develop, build, and deploy cross-platform J2EE applications straight out of the box.

When you download the NetBeans IDE, you get a modular, standards-based development environment with all the key functionality in one download, rather than a series of additional plug-ins. Write, compile, debug and deploy Java programs for the Solaris, Windows, Linux and Macintosh platforms. Download it from:

http://www.netbeans.org

Application Server

The Sun Java System Application Server Platform Edition 8.1 is targeted as the build and runtime environment for the tutorial examples. To build, deploy, and run the examples, you need a copy of the Application Server and the Java 2 Software Development Kit, Standard Edition (J2SE SDK) 1.4.2_06 or higher. If you already have a copy of the J2SE SDK, you can download the Application Server bundled with the IDE from:

http://www.netbeans.info/downloads

Application Server Installation Tips

In the Admin configuration pane of the Application Server installer,

- Select the Don't Prompt for Admin User Name radio button. This will save the user name and password so that you won't need to provide them when performing administrative operations with the IDE. You will still have to provide the user name and password to log in to the Admin Console.
- Note the HTTP port at which the server is installed. This tutorial assumes that you are accepting the default port of 8080. If 8080 is in use during installation and the installer chooses another port or if you decide to change it yourself, you will need to update the configuration files for some of the tutorial examples to reflect the correct port.

Building the Examples

The tutorial examples are distributed with IDE projects. Directions for building the examples are provided in each chapter.

Tutorial Example Directory Structure

To facilitate iterative development and keep application source separate from compiled files, the source code for the tutorial examples is stored in the following structure under each application directory:

- build.xml: build file
- src: Java source of servlets and JavaBeans components; tag libraries
- web: JSP pages and HTML pages, tag files, and images
- nbproject: IDE project files

The build files (build.xml) distributed with the examples contain targets that you can use from the IDE to create a build subdirectory and to copy and compile files into that directory.

Further Information

This tutorial includes the basic information that you need to deploy applications on and administer the Application Server.

For reference information on the tools distributed with the Application Server, see the man pages at http://docs.sun.com/db/doc/817-6092.

See the *Sun Java™ System Application Server Platform Edition 8 Developer's Guide* at http://docs.sun.com/db/doc/817-6087 for information about developer features of the Application Server.

See the *Sun Java™ System Application Server Platform Edition 8 Administration Guide* at http://docs.sun.com/db/doc/817-6088 for information about administering the Application Server.

For information about the PointBase database included with the Application Server see the PointBase web site at www.pointbase.com.

For information about the IDE see the NetBeans site at www.netbeans.org.

How to Print This Tutorial

To print this tutorial, follow these steps:

1. Ensure that Adobe Acrobat Reader is installed on your system.
2. Open the PDF version of this book.
3. Click the printer icon in Adobe Acrobat Reader.

Typographical Conventions

Table 1 lists the typographical conventions used in this tutorial.

Table 1 Typographical Conventions

Font Style	Uses
italic	Emphasis, titles, first occurrence of terms
monospace	URLs, code examples, file names, path names, tool names, application names, programming language keywords, tag, interface, class, method, and field names, properties
italic monospace	Variables in code, file paths, and URLs
<*italic monospace*>	User-selected file path components

Menu selections indicated with the right-arrow character →, for example, First→Second, should be interpreted as: select the First menu, then choose Second from the First submenu.

Feedback

To send comments, broken link reports, errors, suggestions, and questions about this tutorial, please write to nbdocs_feedback@usersguide.netbeans.org.

1 Overview

TODAY, more and more developers want to write distributed transactional applications for the enterprise and thereby leverage the speed, security, and reliability of server-side technology. If you are already working in this area, you know that in the fast-moving and demanding world of e-commerce and information technology, enterprise applications must be designed, built, and produced for less money, with greater speed, and with fewer resources than ever before.

To reduce costs and fast-track application design and development, the Java™ 2 Platform, Enterprise Edition (J2EE™) provides a component-based approach to the design, development, assembly, and deployment of enterprise applications. The J2EE platform offers a multitiered distributed application model, reusable components, a unified security model, flexible transaction control, and web services support through integrated data interchange on Extensible Markup Language (XML)-based open standards and protocols.

Not only can you deliver innovative business solutions to market faster than ever, but also your platform-independent J2EE component-based solutions are not tied to the products and application programming interfaces (APIs) of any one vendor. Vendors and customers enjoy the freedom to choose the products and components that best meet their business and technological requirements.

This tutorial uses examples to describe the features and functionalities available in the J2EE platform version 1.4 for developing enterprise applications. Whether you are a new or an experienced developer, you should find the examples and accompanying text a valuable and accessible knowledge base for creating your own solutions.

If you are new to J2EE enterprise application development, this chapter is a good place to start. Here you will review development basics, learn about the J2EE architecture and APIs, become acquainted with important terms and concepts, and find out how to approach J2EE application programming, assembly, and deployment.

Distributed Multitiered Applications

The J2EE platform uses a distributed multitiered application model for enterprise applications. Application logic is divided into components according to function, and the various application components that make up a J2EE application are installed on different machines depending on the tier in the multitiered J2EE environment to which the application component belongs. Figure 1–1 shows two multitiered J2EE applications divided into the tiers described in the following list. The J2EE application parts shown in Figure 1–1 are presented in J2EE Components (page 3).

- Client-tier components run on the client machine.
- Web-tier components run on the J2EE server.
- Business-tier components run on the J2EE server.
- Enterprise information system (EIS)-tier software runs on the EIS server.

Although a J2EE application can consist of the three or four tiers shown in Figure 1–1, J2EE multitiered applications are generally considered to be three-tiered applications because they are distributed over three locations: client machines, the J2EE server machine, and the database or legacy machines at the back end. Three-tiered applications that run in this way extend the standard two-tiered client and server model by placing a multithreaded application server between the client application and back-end storage.

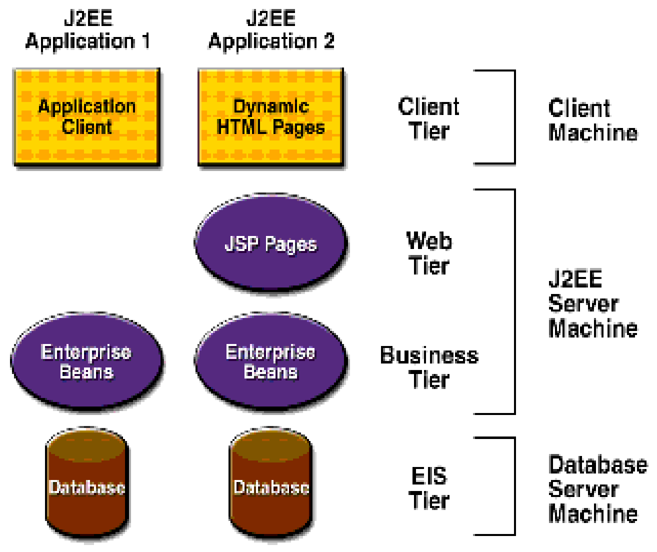


Figure 1–1 Multitiered Applications

J2EE Components

J2EE applications are made up of components. A *J2EE component* is a self-contained functional software unit that is assembled into a J2EE application with its related classes and files and that communicates with other components. The J2EE specification defines the following J2EE components:

- Application clients and applets are components that run on the client.
- Java Servlet and JavaServer Pages™ (JSP™) technology components are web components that run on the server.
- Enterprise JavaBeans™ (EJB™) components (enterprise beans) are business components that run on the server.

J2EE components are written in the Java programming language and are compiled in the same way as any program in the language. The difference between J2EE components and “standard” Java classes is that J2EE components are assembled into a J2EE application, are verified to be well formed and in compliance with the J2EE specification, and are deployed to production, where they are run and managed by the J2EE server.

J2EE Clients

A J2EE client can be a web client or an application client.

Web Clients

A *web client* consists of two parts: (1) dynamic web pages containing various types of markup language (HTML, XML, and so on), which are generated by web components running in the web tier, and (2) a web browser, which renders the pages received from the server.

A web client is sometimes called a *thin client*. Thin clients usually do not query databases, execute complex business rules, or connect to legacy applications. When you use a thin client, such heavyweight operations are off-loaded to enterprise beans executing on the J2EE server, where they can leverage the security, speed, services, and reliability of J2EE server-side technologies.

Applets

A web page received from the web tier can include an embedded applet. An *applet* is a small client application written in the Java programming language that executes in the Java virtual machine installed in the web browser. However, client systems will likely need the Java Plug-in and possibly a security policy file in order for the applet to successfully execute in the web browser.

Web components are the preferred API for creating a web client program because no plug-ins or security policy files are needed on the client systems. Also, web components enable cleaner and more modular application design because they provide a way to separate applications programming from web page design. Personnel involved in web page design thus do not need to understand Java programming language syntax to do their jobs.

Application Clients

An *application client* runs on a client machine and provides a way for users to handle tasks that require a richer user interface than can be provided by a markup language. It typically has a graphical user interface (GUI) created from the Swing or the Abstract Window Toolkit (AWT) API, but a command-line interface is certainly possible.

Application clients directly access enterprise beans running in the business tier. However, if application requirements warrant it, an application client can open an HTTP connection to establish communication with a servlet running in the web tier.

The JavaBeans™ Component Architecture

The server and client tiers might also include components based on the JavaBeans component architecture (JavaBeans components) to manage the data flow between an application client or applet and components running on the J2EE server, or between server components and a database. JavaBeans components are not considered J2EE components by the J2EE specification.

JavaBeans components have properties and have `get` and `set` methods for accessing the properties. JavaBeans components used in this way are typically simple in design and implementation but should conform to the naming and design conventions outlined in the JavaBeans component architecture.

J2EE Server Communications

Figure 1–2 shows the various elements that can make up the client tier. The client communicates with the business tier running on the J2EE server either directly or, as in the case of a client running in a browser, by going through JSP pages or servlets running in the web tier.

Your J2EE application uses a thin browser-based client or thick application client. In deciding which one to use, you should be aware of the trade-offs between keeping functionality on the client and close to the user (thick client) and off-loading as much functionality as possible to the server (thin client). The more functionality you off-load to the server, the easier it is to distribute, deploy, and manage the application; however, keeping more functionality on the client can make for a better perceived user experience.

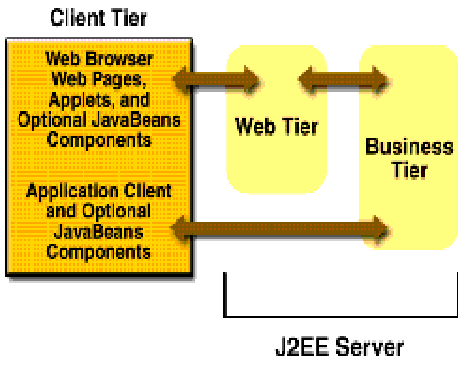

Client Tier
Web Browser Web Pages, Applets, and Optional JavaBeans Components
Application Client and Optional JavaBeans Components
Web Tier
Business Tier
J2EE Server

Figure 1–2 Server Communications

Web Components

J2EE web components are either servlets or pages created using JSP technology (JSP pages). *Servlets* are Java programming language classes that dynamically process requests and construct responses. *JSP pages* are text-based documents that execute as servlets but allow a more natural approach to creating static content.

Static HTML pages and applets are bundled with web components during application assembly but are not considered web components by the J2EE specification. Server-side utility classes can also be bundled with web components and, like HTML pages, are not considered web components.

As shown in Figure 1–3, the web tier, like the client tier, might include a JavaBeans component to manage the user input and send that input to enterprise beans running in the business tier for processing.

Business Components

Business code, which is logic that solves or meets the needs of a particular business domain such as banking, retail, or finance, is handled by enterprise beans running in the business tier. Figure 1–4 shows how an enterprise bean receives data from client programs, processes it (if necessary), and sends it to the enterprise information system tier for storage. An enterprise bean also retrieves data from storage, processes it (if necessary), and sends it back to the client program.

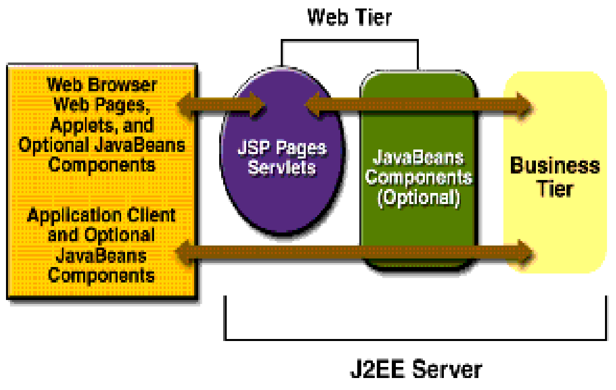

Web Tier
Web Browser Web Pages, Applets, and Optional JavaBeans Components
Application Client and Optional JavaBeans Components
JSP Pages Servlets
JavaBeans Components (Optional)
Business Tier
J2EE Server

Figure 1–3 Web Tier and J2EE Applications

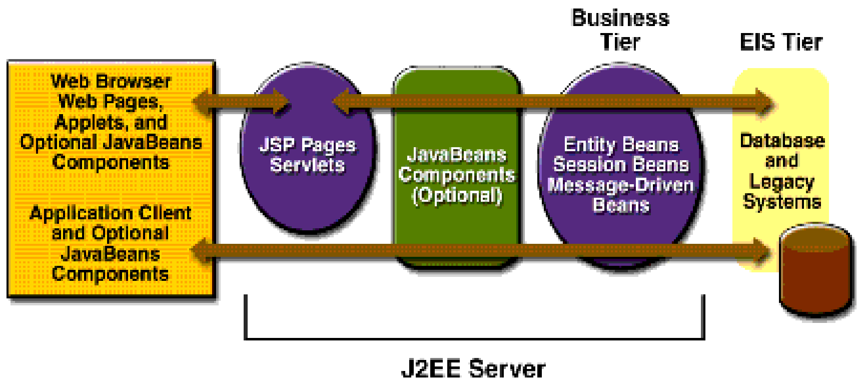

Business Tier
EIS Tier
Web Browser Web Pages, Applets, and Optional JavaBeans Components
Application Client and Optional JavaBeans Components
JSP Pages Servlets
JavaBeans Components (Optional)
Entity Beans Session Beans Message-Driven Beans
Database and Legacy Systems
J2EE Server

Figure 1–4 Business and EIS Tiers

There are three kinds of enterprise beans: session beans, entity beans, and message-driven beans. A *session bean* represents a transient conversation with a client. When the client finishes executing, the session bean and its data are gone. In contrast, an *entity bean* represents persistent data stored in one row of a database table. If the client terminates or if the server shuts down, the underlying services ensure that the entity bean data is saved. A *message-driven bean* combines features of a session bean and a Java Message Service (JMS) message listener, allowing a business component to receive JMS messages asynchronously.

Enterprise Information System Tier

The enterprise information system tier handles EIS software and includes enterprise infrastructure systems such as enterprise resource planning (ERP), mainframe transaction processing, database systems, and other legacy information systems. For example, J2EE application components might need access to enterprise information systems for database connectivity.

J2EE Containers

Normally, thin-client multitiered applications are hard to write because they involve many lines of intricate code to handle transaction and state management, multithreading, resource pooling, and other complex low-level details. The component-based and platform-independent J2EE architecture makes J2EE applications easy to write because business logic is organized into reusable components. In addition, the J2EE server provides underlying services in the form of a container for every component type. Because you do not have to develop these services yourself, you are free to concentrate on solving the business problem at hand.

Container Services

Containers are the interface between a component and the low-level platform-specific functionality that supports the component. Before a web component, enterprise bean, or application client component can be executed, it must be assembled into a J2EE module and deployed into its container.

The assembly process involves specifying container settings for each component in the J2EE application and for the J2EE application itself. Container settings customize the underlying support provided by the J2EE server, including services such as security, transaction management, Java Naming and Directory

Interface™ (JNDI) lookups, and remote connectivity. Here are some of the highlights:

- The J2EE security model lets you configure a web component or enterprise bean so that system resources are accessed only by authorized users.
- The J2EE transaction model lets you specify relationships among methods that make up a single transaction so that all methods in one transaction are treated as a single unit.
- JNDI lookup services provide a unified interface to multiple naming and directory services in the enterprise so that application components can access naming and directory services.
- The J2EE remote connectivity model manages low-level communications between clients and enterprise beans. After an enterprise bean is created, a client invokes methods on it as if it were in the same virtual machine.

Because the J2EE architecture provides configurable services, application components within the same J2EE application can behave differently based on where they are deployed. For example, an enterprise bean can have security settings that allow it a certain level of access to database data in one production environment and another level of database access in another production environment.

The container also manages nonconfigurable services such as enterprise bean and servlet life cycles, database connection resource pooling, data persistence, and access to the J2EE platform APIs described in section J2EE 1.4 APIs (page 15). Although data persistence is a nonconfigurable service, the J2EE architecture lets you override container-managed persistence by including the appropriate code in your enterprise bean implementation when you want more control than the default container-managed persistence provides. For example, you might use bean-managed persistence to implement your own finder (search) methods or to create a customized database cache.

Container Types

The deployment process installs J2EE application components in the J2EE containers illustrated in Figure 1–5.

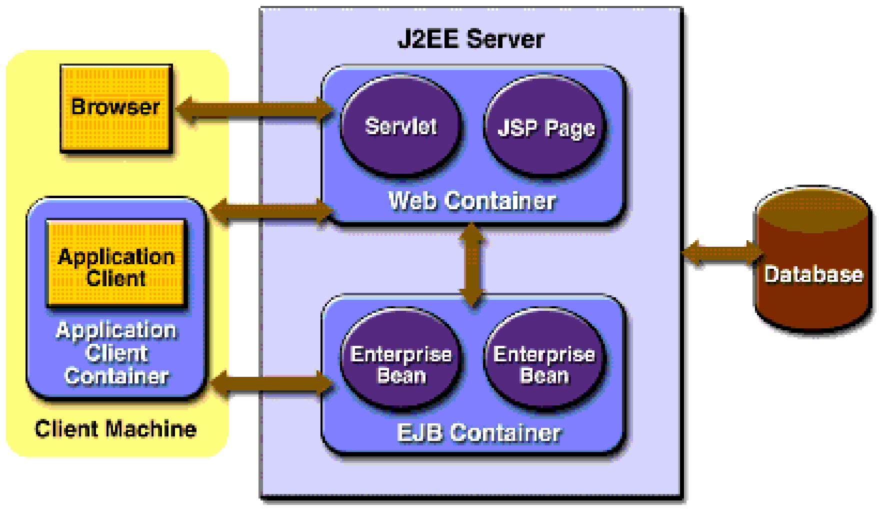


Figure 1–5 J2EE Server and Containers

J2EE server
The runtime portion of a J2EE product. A J2EE server provides EJB and web containers.

Enterprise JavaBeans (EJB) container
Manages the execution of enterprise beans for J2EE applications. Enterprise beans and their container run on the J2EE server.

Web container
Manages the execution of JSP page and servlet components for J2EE applications. web components and their container run on the J2EE server.

Application client container
Manages the execution of application client components. Application clients and their container run on the client.

Applet container
Manages the execution of applets. Consists of a web browser and Java Plug-in running on the client together.

Web Services Support

Web services are web-based enterprise applications that use open, XML-based standards and transport protocols to exchange data with calling clients. The J2EE platform provides the XML APIs and tools you need to quickly design, develop, test, and deploy web services and clients that fully interoperate with other web services and clients running on Java-based or non-Java-based platforms.

To write web services and clients with the J2EE XML APIs, all you do is pass parameter data to the method calls and process the data returned; or for document-oriented web services, you send documents containing the service data back and forth. No low-level programming is needed because the XML API implementations do the work of translating the application data to and from an XML-based data stream that is sent over the standardized XML-based transport protocols. These XML-based standards and protocols are introduced in the following sections.

The translation of data to a standardized XML-based data stream is what makes web services and clients written with the J2EE XML APIs fully interoperable. This does not necessarily mean that the data being transported includes XML tags because the transported data can itself be plain text, XML data, or any kind of binary data such as audio, video, maps, program files, computer-aided design (CAD) documents and the like. The next section introduces XML and explains how parties doing business can use XML tags and schemas to exchange data in a meaningful way.

XML

XML is a cross-platform, extensible, text-based standard for representing data. When XML data is exchanged between parties, the parties are free to create their own tags to describe the data, set up schemas to specify which tags can be used in a particular kind of XML document, and use XML stylesheets to manage the display and handling of the data.

For example, a web service can use XML and a schema to produce price lists, and companies that receive the price lists and schema can have their own stylesheets to handle the data in a way that best suits their needs. Here are examples:

- One company might put XML pricing information through a program to translate the XML to HTML so that it can post the price lists to its intranet.
- A partner company might put the XML pricing information through a tool to create a marketing presentation.
- Another company might read the XML pricing information into an application for processing.

SOAP Transport Protocol

Client requests and web service responses are transmitted as Simple Object Access Protocol (SOAP) messages over HTTP to enable a completely interoperable exchange between clients and web services, all running on different platforms and at various locations on the Internet. HTTP is a familiar request-and response standard for sending messages over the Internet, and SOAP is an XML-based protocol that follows the HTTP request-and-response model.

The SOAP portion of a transported message handles the following:

- Defines an XML-based envelope to describe what is in the message and how to process the message
- Includes XML-based encoding rules to express instances of application-defined data types within the message
- Defines an XML-based convention for representing the request to the remote service and the resulting response

WSDL Standard Format

The Web Services Description Language (WSDL) is a standardized XML format for describing network services. The description includes the name of the service, the location of the service, and ways to communicate with the service. WSDL service descriptions can be stored in UDDI registries or published on the web (or both). The Sun Java System Application Server Platform Edition 8 provides a tool for generating the WSDL specification of a web service that uses remote procedure calls to communicate with clients.

UDDI and ebXML Standard Formats

Other XML-based standards, such as Universal Description, Discovery and Integration (UDDI) and ebXML, make it possible for businesses to publish information on the Internet about their products and web services, where the information can be readily and globally accessed by clients who want to do business.

Packaging Applications

A J2EE application is delivered in an Enterprise Archive (EAR) file, a standard Java Archive (JAR) file with an `.ear` extension. Using EAR files and modules makes it possible to assemble a number of different J2EE applications using some of the same components. No extra coding is needed; it is only a matter of assembling (or packaging) various J2EE modules into J2EE EAR files.

An EAR file (see Figure 1–6) contains J2EE modules and deployment descriptors. A *deployment descriptor* is an XML document with an `.xml` extension that describes the deployment settings of an application, a module, or a component. Because deployment descriptor information is declarative, it can be changed without the need to modify the source code. At runtime, the J2EE server reads the deployment descriptor and acts upon the application, module, or component accordingly.

There are two types of deployment descriptors: J2EE and runtime. A *J2EE deployment descriptor* is defined by a J2EE specification and can be used to configure deployment settings on any J2EE-compliant implementation. A *runtime deployment descriptor* is used to configure J2EE implementation-specific parameters. For example, the Sun Java System Application Server Platform Edition 8 runtime deployment descriptor contains information such as the context root of a web application, the mapping of portable names of an application's resources to the server's resources, and Application Server implementation-specific parameters, such as caching directives. The Application Server runtime deployment descriptors are named `sun-`*moduleType*`.xml` and are located in the same directory as the J2EE deployment descriptor.

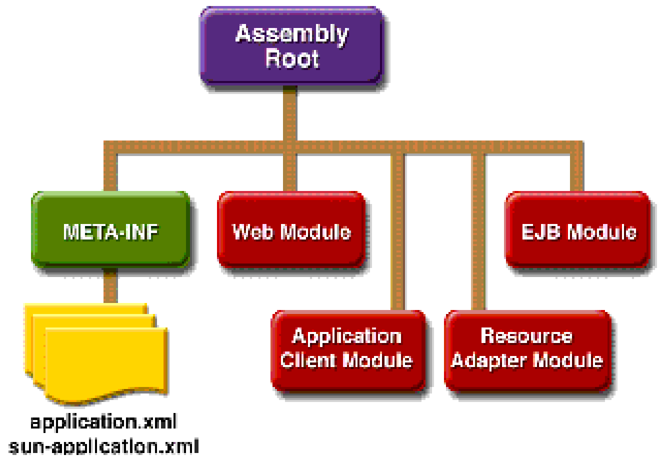


Figure 1–6 EAR File Structure

A *J2EE module* consists of one or more J2EE components for the same container type and one component deployment descriptor of that type. An enterprise bean module deployment descriptor, for example, declares transaction attributes and security authorizations for an enterprise bean. A J2EE module without an application deployment descriptor can be deployed as a *stand-alone* module. The four types of J2EE modules are as follows:

- EJB modules, which contain class files for enterprise beans and an EJB deployment descriptor. EJB modules are packaged as JAR files with a `.jar` extension.
- Web modules, which contain servlet class files, JSP files, supporting class files, GIF and HTML files, and a web application deployment descriptor. Web modules are packaged as JAR files with a `.war` (web archive) extension.
- Application client modules, which contain class files and an application client deployment descriptor. Application client modules are packaged as JAR files with a `.jar` extension.
- Resource adapter modules, which contain all Java interfaces, classes, native libraries, and other documentation, along with the resource adapter deployment descriptor. Together, these implement the Connector architecture (see J2EE Connector Architecture, page 19) for a particular EIS. Resource adapter modules are packaged as JAR files with an `.rar` (resource adapter archive) extension.

J2EE 1.4 APIs

Figure 1–7 illustrates the availability of the J2EE 1.4 platform APIs in each J2EE container type. The following sections give a brief summary of the technologies required by the J2EE platform and the J2SE enterprise APIs that would be used in J2EE applications.

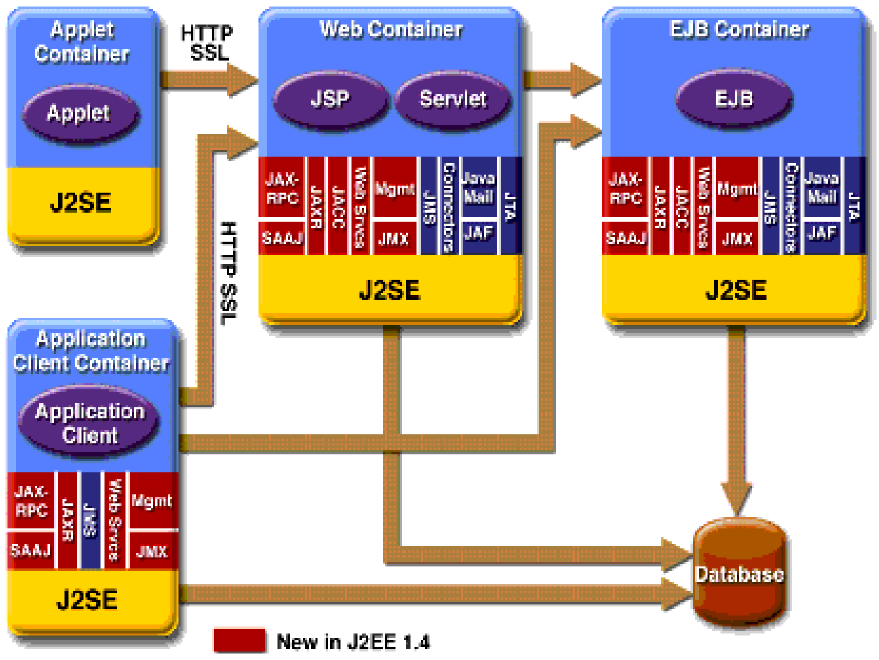


Figure 1–7 J2EE Platform APIs

Enterprise JavaBeans Technology

An Enterprise JavaBeans™ (EJB™) component, or *enterprise bean*, is a body of code having fields and methods to implement modules of business logic. You can think of an enterprise bean as a building block that can be used alone or with other enterprise beans to execute business logic on the J2EE server.

As mentioned earlier, there are three kinds of enterprise beans: session beans, entity beans, and message-driven beans. Enterprise beans often interact with databases. One of the benefits of entity beans is that you do not have to write any SQL code or use the JDBC™ API (see JDBC API, page 19) directly to perform database access operations; the EJB container handles this for you. However, if you override the default container-managed persistence for any reason, you will need to use the JDBC API. Also, if you choose to have a session bean access the database, you must use the JDBC API.

Java Servlet Technology

Java servlet technology lets you define HTTP-specific servlet classes. A servlet class extends the capabilities of servers that host applications that are accessed by way of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers.

JavaServer Pages Technology

JavaServer Pages™ (JSP™) technology lets you put snippets of servlet code directly into a text-based document. A JSP page is a text-based document that contains two types of text: static data (which can be expressed in any text-based format such as HTML, WML, and XML) and JSP elements, which determine how the page constructs dynamic content.

Java Message Service API

The Java Message Service (JMS) API is a messaging standard that allows J2EE application components to create, send, receive, and read messages. It enables distributed communication that is loosely coupled, reliable, and asynchronous.

Java Transaction API

The Java Transaction API (JTA) provides a standard interface for demarcating transactions. The J2EE architecture provides a default auto commit to handle transaction commits and rollbacks. An *auto commit* means that any other applications that are viewing data will see the updated data after each database read or write operation. However, if your application performs two separate database access operations that depend on each other, you will want to use the JTA API to demarcate where the entire transaction, including both operations, begins, rolls back, and commits.

JavaMail API

J2EE applications use the JavaMail™ API to send email notifications. The JavaMail API has two parts: an application-level interface used by the application components to send mail, and a service provider interface. The J2EE platform includes JavaMail with a service provider that allows application components to send Internet mail.

JavaBeans Activation Framework

The JavaBeans Activation Framework (JAF) is included because JavaMail uses it. JAF provides standard services to determine the type of an arbitrary piece of data, encapsulate access to it, discover the operations available on it, and create the appropriate JavaBeans component to perform those operations.

Java API for XML Processing

The Java API for XML Processing (JAXP) supports the processing of XML documents using Document Object Model (DOM), Simple API for XML (SAX), and Extensible Stylesheet Language Transformations (XSLT). JAXP enables applications to parse and transform XML documents independent of a particular XML processing implementation.

JAXP also provides namespace support, which lets you work with schemas that might otherwise have naming conflicts. Designed to be flexible, JAXP lets you use any XML-compliant parser or XSL processor from within your application and supports the W3C schema. You can find information on the W3C schema at this URL: `http://www.w3.org/XML/Schema`.

Java API for XML-Based RPC

The Java API for XML-based RPC (JAX-RPC) uses the SOAP standard and HTTP, so client programs can make XML-based remote procedure calls (RPCs) over the Internet. JAX-RPC also supports WSDL, so you can import and export WSDL documents. With JAX-RPC and a WSDL, you can easily interoperate with clients and services running on Java-based or non-Java-based platforms such as .NET. For example, based on the WSDL document, a Visual Basic .NET client can be configured to use a web service implemented in Java technology, or a web service can be configured to recognize a Visual Basic .NET client.

JAX-RPC relies on the HTTP transport protocol. Taking that a step further, JAX-RPC lets you create service applications that combine HTTP with a Java technology version of the Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocols to establish basic or mutual authentication. SSL and TLS ensure message integrity by providing data encryp

J2EE Connector Architecture

The J2EE Connector architecture is used by J2EE tools vendors and system integrators to create resource adapters that support access to enterprise information systems that can be plugged in to any J2EE product. A *resource adapter* is a software component that allows J2EE application components to access and interact with the underlying resource manager of the EIS. Because a resource adapter is specific to its resource manager, typically there is a different resource adapter for each type of database or enterprise information system.

The J2EE Connector architecture also provides a performance-oriented, secure, scalable, and message-based transactional integration of J2EE-based web services with existing EISs that can be either synchronous or asynchronous. Existing applications and EISs integrated through the J2EE Connector architecture into the J2EE platform can be exposed as XML-based web services by using JAX-RPC and J2EE component models. Thus JAX-RPC and the J2EE Connector architecture are complementary technologies for enterprise application integration (EAI) and end-to-end business integration.

JDBC API

The JDBC API lets you invoke SQL commands from Java programing language methods. You use the JDBC API in an enterprise bean when you override the default container-managed persistence or have a session bean access the database. With container-managed persistence, database access operations are handled by the container, and your enterprise bean implementation contains no JDBC code or SQL commands. You can also use the JDBC API from a servlet or a JSP page to access the database directly without going through an enterprise bean.

The JDBC API has two parts: an application-level interface used by the application components to access a database, and a service provider interface to attach a JDBC driver to the J2EE platform.

Java Naming and Directory Interface

The Java Naming and Directory Interface™ (JNDI) provides naming and directory functionality. It provides applications with methods for performing standard directory operations, such as associating attributes with objects and searching for objects using their attributes. Using JNDI, a J2EE application can store and retrieve any type of named Java object.

J2EE naming services provide application clients, enterprise beans, and web components with access to a JNDI naming environment. A *naming environment* allows a component to be customized without the need to access or change the component's source code. A container implements the component's environment and provides it to the component as a JNDI *naming context*.

A J2EE component locates its environment naming context using JNDI interfaces. A component creates a `javax.naming.InitialContext` object and looks up the environment naming context in `InitialContext` under the name `java:comp/env`. A component's naming environment is stored directly in the environment naming context or in any of its direct or indirect subcontexts.

A J2EE component can access named system-provided and user-defined objects. The names of system-provided objects, such as JTA `UserTransaction` objects, are stored in the environment naming context, `java:comp/env`. The J2EE platform allows a component to name user-defined objects, such as enterprise beans, environment entries, JDBC `DataSource` objects, and message connections. An object should be named within a subcontext of the naming environment according to the type of the object. For example, enterprise beans are named within the subcontext `java:comp/env/ejb`, and JDBC `DataSource` references in the subcontext `java:comp/env/jdbc`.

Because JNDI is independent of any specific implementation, applications can use JNDI to access multiple naming and directory services, including existing naming and directory services such as LDAP, NDS, DNS, and NIS. This allows J2EE applications to coexist with legacy applications and systems. For more information on JNDI, see *The JNDI Tutorial*:

```
http://java.sun.com/products/jndi/tutorial/index.html
```

Java Authentication and Authorization Service

The Java Authentication and Authorization Service (JAAS) provides a way for a J2EE application to authenticate and authorize a specific user or group of users to run it.

JAAS is a Java programing language version of the standard Pluggable Authentication Module (PAM) framework, which extends the Java 2 Platform security architecture to support user-based authorization.

Simplified Systems Integration

The J2EE platform is a platform-independent, full systems integration solution that creates an open marketplace in which every vendor can sell to every customer. Such a marketplace encourages vendors to compete, not by trying to lock customers into their technologies but instead by trying to outdo each other in providing products and services that benefit customers, such as better performance, better tools, or better customer support.

The J2EE APIs enable systems and applications integration through the following:

- Unified application model across tiers with enterprise beans
- Simplified request-and-response mechanism with JSP pages and servlets
- Reliable security model with JAAS
- XML-based data interchange integration with JAXP, SAAJ, and JAX-RPC
- Simplified interoperability with the J2EE Connector architecture
- Easy database connectivity with the JDBC API
- Enterprise application integration with message-driven beans and JMS, JTA, and JNDI

You can learn more about using the J2EE platform to build integrated business systems by reading *J2EE Technology in Practice*, by Rick Cattell and Jim Inscore (Addison-Wesley, 2001):

```
http://java.sun.com/j2ee/inpractice/aboutthebook.html
```

Sun Java System Application Server Platform Edition 8

The Sun Java System Application Server Platform Edition 8 is a fully compliant implementation of the J2EE 1.4 platform. In addition to supporting all the APIs described in the previous sections, the Application Server includes a number of

J2EE technologies and tools that are not part of the J2EE 1.4 platform but are provided as a convenience to the developer.

This section briefly summarizes the technologies and tools that make up the Application Server, and instructions for starting and stopping the Application Server, starting the Admin Console, starting `deploytool`, and starting and stopping the PointBase database server. Other chapters explain how to use the remaining tools.

Technologies

The Application Server includes two user interface technologies—JavaServer Pages Standard Tag Library and JavaServer™ Faces—that are built on and used in conjunction with the J2EE 1.4 platform technologies Java servlet and JavaServer Pages.

JavaServer Pages Standard Tag Library

The JavaServer Pages Standard Tag Library (JSTL) encapsulates core functionality common to many JSP applications. Instead of mixing tags from numerous vendors in your JSP applications, you employ a single, standard set of tags. This standardization allows you to deploy your applications on any JSP container that supports JSTL and makes it more likely that the implementation of the tags is optimized.

JSTL has iterator and conditional tags for handling flow control, tags for manipulating XML documents, internationalization tags, tags for accessing databases using SQL, and commonly used functions.

JavaServer Faces

JavaServer Faces technology is a user interface framework for building web applications. The main components of JavaServer Faces technology are as follows:

- A GUI component framework.
- A flexible model for rendering components in different kinds of HTML or different markup languages and technologies. A `Renderer` object generates the markup to render the component and converts the data stored in a model object to types that can be represented in a view.

- A standard `RenderKit` for generating HTML/4.01 markup.

The following features support the GUI components:

- Input validation
- Event handling
- Data conversion between model objects and components
- Managed model object creation
- Page navigation configuration

All this functionality is available via standard Java APIs and XML-based configuration files.

Tools

The Application Server contains the tools listed in Table 1–1. All can be used from the IDE. Basic usage information for many of the tools appears throughout the tutorial. For detailed information, see the online help in the GUI tools and the man pages at `http://docs.sun.com/db/doc/817-6092` for the command-line tools.

Table 1–1 Application Server Tools

Component	**Description**
Admin Console	A web-based GUI Application Server administration utility. Used to stop the Application Server and manage users, resources, and applications.
PointBase database	An evaluation copy of the PointBase database server.
verifier	A command-line tool to validate J2EE deployment descriptors.
wscompile	A command-line tool to generate stubs, ties, serializers, and WSDL files used in JAX-RPC clients and services.
wsdeploy	A command-line tool to generate implementation-specific, ready-to-deploy WAR files for web service applications that use JAX-RPC.

Registering the Application Server

To register the Application Server, you can use the IDE. Note that if you downloaded and installed the version of the IDE that comes bundled with the Application Server, you do not have to perform this step. The IDE knows the location of the bundled Application Server.

1. In the IDE, choose Tools→Server Manager from the main window.
2. Click Add Server. Select Sun Java Systems Application Server 8.1 and give a name to the instance. Then click Next.
3. Specify the installation directory of the application server (for example, `C:\Sun\Appserver`) and click Next.
4. Select the location of a local instance of the application server from the Location combo box.
5. Optionally, enter your administrator username and password. If you do not want to store the username and password in your IDE user directory, you can leave these fields blank. The IDE will prompt you every time it needs the information. Note that the default `admin` password is `adminadmin`.

Starting and Stopping the Application Server

To start and stop the Application Server, you can use the IDE. To start the Application Server, open the IDE, go to the Runtime window (Ctrl-5), expand the Servers node, right-click the Application Server's node, and choose Start/Stop Server. In the Server Status dialog box, click Start Server.

A *domain* is a set of one or more Application Server instances managed by one administration server. Associated with a domain are the following:

- The Application Server's port number. The default is 8080.
- The administration server's port number. The default is 4848.
- An administration user name and password.

You specify these values when you install the Application Server. The examples in this tutorial assume that you choose the default ports.

With no arguments, the IDE initiates the default domain, which is domain1. The `--verbose` flag causes all logging and debugging output to appear on the terminal window or command prompt (it will also go into the server log, which is located in <*J2EE_HOME*>`/domains/domain1/logs/server.log`).

After the server has completed its startup sequence, you will see the following output in the IDE's Output window:

```
Domain domain1 started.
```

To stop the Application Server, click Stop Server in the Server Status dialog box.When the server has stopped you will see the following output in the IDE's Output window:

```
Domain domain1 stopped.
```

Starting the Admin Console

To administer the Application Server and manage users, resources, and J2EE applications, you use the Admin Console tool. The Application Server must be running before you invoke the Admin Console. To start the Admin Console, open the IDE, go to the Runtime window (Ctrl-5), expand the Servers node, right-click the node for the Application Server, and choose View Admin Console.

Starting and Stopping the PointBase Database Server

The Application Server includes an evaluation copy of the PointBase database.

To start the PointBase database server, open the IDE and choose Tools→Pointbase Database→Start Local Pointbase Database from the main menu.

For information about the PointBase database included with the Application Server see the PointBase web site at `www.pointbase.com`.

Debugging J2EE Applications

This section describes how to determine what is causing an error in your application deployment or execution.

Using the Server Log

One way to debug applications is to look at the server log in *<J2EE_HOME>*`/domains/domain1/logs/server.log`. The log contains output from the Application Server and your applications. You can log messages from any Java class in your application with `System.out.println` and the Java Logging APIs (documented at `http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/index.html`) and from web components with the `ServletContext.log` method.

You can also view the Server Log in the IDE. Open the IDE, go to the Runtime window (Ctrl-5), expand the Servers node, right-click the node for the Application Server, and choose View Server Log.

If you start the Application Server with the `--verbose` flag, all logging and debugging output will appear on the terminal window or command prompt and the server log. If you start the Application Server in the background, debugging information is only available in the log. You can view the server log with a text editor or with the Admin Console log viewer. To use the log viewer:

1. Select the Application Server node.
2. Select the Logging tab.
3. Click the Open Log Viewer button. The log viewer will open and display the last 40 entries.

If you wish to display other entries:

1. Click the Modify Search button.
2. Specify any constraints on the entries you want to see.
3. Click the Search button at the bottom of the log viewer.

Using the NetBeans Debugger

The IDE uses the Sun Microsystems JPDA debugger to debug your programs. When you start a debugging session, all of the relevant debugger windows appear automatically at the bottom of your screen. You can debug an entire project, any executable class, and any JUnit tests. The IDE also lets you debug applications that are running on a remote machine by attaching the debugger to the application process.

When you run or debug web applications, JSP pages, or servlets, you can also use the HTTP Monitor to monitor data flow. The HTTP Monitor appears by default. The HTTP Monitor gathers data about HTTP requests that the servlet engine processes. For each HTTP request that the engine processes, the monitor records data about the incoming request, the data states maintained on the server, and the servlet context. You can view data, store data for future sessions, and replay and edit previous requests.

When you start a debugging session in the IDE, the IDE compiles the files that you are debugging, runs them in debug mode, and displays debugger output in the Debugger windows. To start a debugging session, select the file that you want to debug and choose one of the following commands from the Run menu:

- Debug Main Project (F5). Runs the main project until the first breakpoint is encountered.
- Step Into (F7). Starts running the main project's main class and stops at the first executable statement.
- Run to Cursor (F4). Starts a debugging session, runs the application to the cursor location in the Source Editor, and pauses the application.

4.

2

Building Web Services with JAX-RPC

JAX-RPC stands for Java API for XML-based RPC. JAX-RPC is a technology for building web services and clients that use *remote procedure calls* (RPC) and XML. Often used in a distributed client-server model, an RPC mechanism enables clients to execute procedures on other systems.

In JAX-RPC, a remote procedure call is represented by an XML-based protocol such as SOAP. The SOAP specification defines the envelope structure, encoding rules, and conventions for representing remote procedure calls and responses. These calls and responses are transmitted as SOAP messages (XML files) over HTTP.

Although SOAP messages are complex, the JAX-RPC API hides this complexity from the application developer. On the server side, the developer specifies the remote procedures by defining methods in an interface written in the Java programming language. The developer also codes one or more classes that implement those methods. Client programs are also easy to code. A client creates a proxy (a local object representing the service) and then simply invokes methods on the proxy. With JAX-RPC, the developer does not generate or parse SOAP messages. It is the JAX-RPC runtime system that converts the API calls and responses to and from SOAP messages.

With JAX-RPC, clients and web services have a big advantage: the platform independence of the Java programming language. In addition, JAX-RPC is not restrictive: a JAX-RPC client can access a web service that is not running on the

Java platform, and vice versa. This flexibility is possible because JAX-RPC uses technologies defined by the World Wide Web Consortium (W3C): HTTP, SOAP, and the Web Service Description Language (WSDL). WSDL specifies an XML format for describing a service as a set of endpoints operating on messages.

Setting the Port

Several files in the JAX-RPC examples depend on the port that you specified when you installed the Sun Java System Application Server Platform Edition 8.1. The tutorial examples assume that the server runs on the default port, 8080. If you have changed the port, you must update the port number in the following files before building and running the JAX-RPC examples:

- *<INSTALL>*`/j2eetutorial14/examples/jaxrpc/staticstub/src/conf/Hello-staticclient-config.xml`
- *<INSTALL>*`/j2eetutorial14/examples/jaxrpc/dynamicproxy/src/conf/Hello-dynamicclient-config.xml`
- *<INSTALL>*`/j2eetutorial14/examples/jaxrpc/webclient/nbproject/project.xml`

As explained earlier, you need to register the Sun Java System Application Server Platform Edition 8.1 in the IDE before going any further with this chapter.

Creating a Simple Web Service and Client with JAX-RPC

This section shows how to build, deploy, and consume a simple web service. You will learn about two types of web service clients in this section. Both are static-stub clients, which means that they call the web service through a *stub*, a local object that acts as a proxy for the remote service. The difference between the two clients in this section is that one is portable, because it adheres to the J2EE 1.4 specification, while the other is not. A later section, Web Service Clients (page 43), provides examples of additional JAX-RPC clients that access the service. The code for the service is in *<INSTALL>*`/j2eetutorial14/examples/jaxrpc/helloservice`. The portable client is in *<INSTALL>*`/j2eetutorial14/examples/jaxrpc/webclient` and the client that is implementation-specific is in *<INSTALL>*`/j2eetutorial14/examples/jaxrpc/staticstub`.

Figure 2–1 illustrates how JAX-RPC technology manages communication between a web service and client.

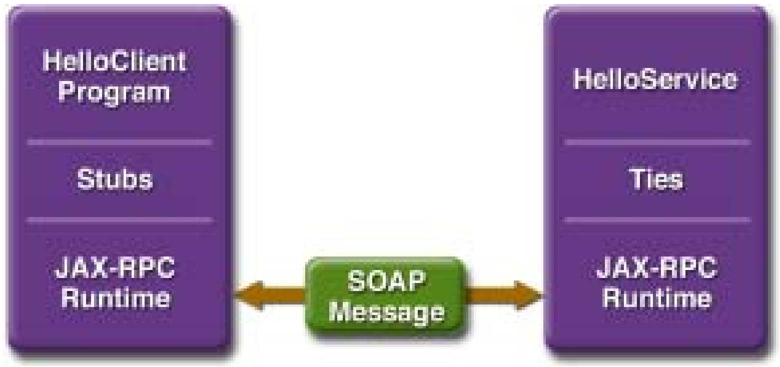


Figure 2–1 Communication Between a JAX-RPC Web Service and a Client

The starting point for developing a JAX-RPC web service is the service endpoint interface. A *service endpoint interface* (SEI) is a Java interface that declares the methods that a client can invoke on the service.

You run the `wscompile` tool from the IDE to process the SEI and two configuration files. Doing so generates the WSDL specification of the web service and the stubs that connect a web service client to the JAX-RPC runtime. For reference documentation on `wscompile`, see the Application Server man pages at `http://docs.sun.com/db/doc/817-6092`.

Together, the `wscompile` tool, the IDE, and the Application Server provide the Application Server's implementation of JAX-RPC.

These are the basic steps for creating the web service and client in the IDE:

1. Generate the SEI, the implementation class, and the interface configuration file. Code the implementation class.
2. Compile the SEI and implementation class. During this step, the `wscompile` tool is called from the IDE to generate the files required to deploy the service.
3. Package and deploy the WAR file. The tie classes (which are used to communicate with clients) are generated by the Application Server during deployment.

4. Generate and code the client class and WSDL configuration file.
5. Compile the client class. During this step, the `wscompile` tool is called from the IDE to generate and compile the stub files.
6. Package and run the client class.

The sections that follow cover these steps in greater detail.

Generating and Coding the Service Endpoint Interface and Implementation Class

In this example, the service endpoint interface declares a single method named `sayHello`. This method returns a string that is the concatenation of the string `Hello` with the method parameter.

A service endpoint interface must conform to a few rules:

- It extends the `java.rmi.Remote` interface.
- It must not have constant declarations, such as `public final static`.
- The methods must throw the `java.rmi.RemoteException` or one of its subclasses. (The methods may also throw service-specific exceptions.)
- Method parameters and return types must be supported JAX-RPC types (see Types Supported by JAX-RPC, page 42).

To generate the SEI, the implementation class, and the interface configuration file, use the IDE as follows:

1. Choose File→New Project. In the Categories tree, choose Web. Under Projects, choose Web Application. Click Next.
2. In the Project Name field, type `helloservice`. In the Project Location field, browse to the location where all your projects are stored. In the Server field, make sure that the Sun Java System Application Server Platform Edition is selected. (If the Sun Java System Application Server is not available in the Server field, you need to register it in the IDE. Choose Tools→Server Manager to do so.) Click Finish.
3. Right-click `helloservice` in the Projects window. Choose New→Web Service. In the Web Service Name field, type `Hello`. In the Package field, type `helloservice`. Click Finish.
4. Expand the Web Services node in the Projects window, right-click the `Hello` node, and choose Add Operation. In the Name field, type `say-Hello.` In the Return Type field, choose `String.`

5. Click Add. Leave the Type as `String.` In the Name field, type `s`. Click OK and then click OK again.
6. Add `public String message = "Hello ";` below the `HelloImpl` class declaration.
7. Implement the `sayHello` operation by replacing the default `return null` with `return message + s.`

Expand the Source Packages node in the Projects window. Then expand the `helloservice` package node. In this example, the service endpoint interface that the IDE generates for you is named `HelloSEI`. Double-click it in the Projects window to view it in the Source Editor:

```
package helloservice;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HelloSEI extends Remote {
    public String sayHello(String s) throws RemoteException;
}
```

In addition to the interface, you'll need the class that implements the interface. In this example, the implementation class is called `HelloImpl`. Double-click it in the Projects window to view it in the Source Editor:

```
package helloservice;

public class HelloImpl implements HelloSEI {

    public String message ="Hello";

    public String sayHello(String s) {
        return message + s;
    }
}
```

Building the Service

To build the `helloservice`, right-click the node in the Projects window and choose Build Project. The Build Project command executes several subtasks in your Ant build script, the most important of which are the following:

- `compile`
- `Hello_wscompile`

- dist

The compile Task

This task compiles `HelloSEI.java` and `HelloImpl.java`, writing the class files to the `build/web/WEB-INF/classes` subdirectory, which you can view in the Files window (Ctrl-2).

The Hello_wscompile Task

The `Hello_wscompile` task runs `wscompile`, which creates the WSDL and mapping files. You can view them by going to the `build/web/WEB-INF` subdirectory and the `build/web/WEB-INF/wsdl` subdirectory in the Files window. The WSDL file describes the web service and is used to generate the client stubs for Static Stub Clients. The mapping file contains information that correlates the mapping between the Java interfaces and the WSDL definition. It is meant to be portable so that any J2EE-compliant deployment tool can use this information, along with the WSDL file and the Java interfaces, to generate stubs and ties for the deployed web services.

The files created in this example are `Hello.wsdl` and `Hello-mapping.xml`. The `Hello_wscompile` task runs `wscompile` with the following main arguments:

```
wscompile
  define="true"
  nonClassDir="${build.web.dir.real}/WEB-INF/wsdl"
  mapping="${build.web.dir.real}/WEB-INF/${Hello.mapping}"
  config="${src.dir}/${Hello.config.name}"
  features="${wscompile.service.Hello.features}"
  sourceBase="${build.generated.dir}/wsservice"
```

The `define` option instructs `wscompile` to create WSDL and mapping files. The `mapping` option specifies the mapping file name. The other options specify various properties that are set in the `nbproject/project.properties` file. The `wscompile` tool reads an interface configuration file that specifies information about the SEI. In this example, the configuration file is named `Hello-config.xml` and contains the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
<service
       name="Hello" targetNamespace="urn:Hello/wsdl"
```

```
          typeNamespace="urn:Hello/types"
          packageName="helloservice">
     <interface name="helloservice.HelloSEI"
             servantName='helloservice.HelloImpl'</interface>
   </service>
   </configuration>
```

This configuration file tells `wscompile` to create a WSDL file named `Hello.wsdl` with the following information:

- The service name is `Hello`.
- The WSDL target is `urn:Hello/wsdl` and the type namespace is urn:Hello/types. The choice for what to use for the namespaces is up to you. The role of the namespaces is similar to the use of Java package names—to distinguish names that might otherwise conflict. For example, a company can decide that all its Java code should be in the package `com.wombat.*`. Similarly, it can also decide to use the namespace `http://wombat.com`.
- The SEI is `helloservice.HelloSEI`.

The `packageName` attribute instructs `wscompile` to put the service classes into the `helloservice` package.

The dist Task

This task packages the service and the deployment descriptor into a WAR file in the `dist` folder, which you can view in the Files window.

Specifying the Endpoint Address

To access `helloservice`, the tutorial clients will specify this service endpoint address URI:

```
   http://localhost:8080/helloservice/Hello
```

The `/helloservice` string is the context root of the servlet that implements the service. The `/Hello` string is the servlet alias. You already set the context root when you created the web application above. To specify the endpoint address, set the alias as follows:

1. Right-click the project node, choose Properties, and then click Run in the Project Properties dialog box.

2. In the Relative URL field, type `/Hello`.

Deploying the Service

In the IDE, perform these steps:

1. In the main menu, choose Tools→Setup Wizard. Select your favorite browser in the Web Browser drop-down and click Finish.
2. In the Projects window, right-click `helloservice` and choose Run Project.

You can view the WSDL file of the deployed service by requesting the URL `http://localhost:8080/helloservice/Hello?WSDL` in a web browser. Now you are ready to create a client that accesses this service.

Undeploying the Service

At this point in the tutorial, do not undeploy the service. When you are finished with this example, you can undeploy the service by expanding the Servers node in the Runtime window, then the node for the server, then right-click the node for the service, and choose Undeploy.

Static Stub Clients

You will create a stand-alone program that calls the `sayHello` method of the `helloservice`. It makes this call through a *stub*, a local object that acts as a proxy for the remote service. Because the stub is created by `wscompile` at development time (as opposed to runtime), it is usually called a *static stub*. You can run `wscompile` from the IDE to generate the stub in one of two ways:

- J2EE Container-Generated
  This stub is created by the server, using information gleaned from deployment descriptors generated in the IDE.
- IDE-Generated
  This stub is created manually in the IDE. As a result, it is implementation-specific, as discussed later in this chapter.

J2EE Container-Generated Static Stub Client

To generate the static stub client, use the IDE as follows:

1. Choose File→New Project. Under Categories, choose Web. Under Projects, choose Web Application. Click Next.
2. In the Project Name field, type `HelloClientProject`. In the Project Location field, browse to the location where all your projects are stored. In the Server field, make sure that the Sun Java System Application Server Platform Edition 8.1 is selected. Click Finish.
3. Right-click `HelloClientProject` in the Projects window. Choose New→Web Service Client. In the WSDL URL field, specify the URL to the web service:

```
http://localhost:8080/helloservice/Hello?WSDL
```

4. Click Retrieve WSDL to test the location. If the WSDL name is returned, the test has succeeded. In the Package field, type `helloclientservice`. In the Web Service Client Type list, choose J2EE Container- Generated Static Stub.
5. Right-click the `HelloClientProject` node and choose New→Servlet. In the Name field, type `HelloServlet.` In the Package field, type `webclient.` Click Next and click Finish.
6. Right-click within the `processRequest` method and choose Web Service Client Resources→Call Web Service Operation. Choose the `sayHello` operation and click OK. Now fill out the skeleton code so that the content of its `<body>` tags looks as follows:

```
out.println("<body>");
String username = request.getParameter("username");
if (username != null && username.length() > 0) {
  try {
     out.println("<img src=\"duke.waving.gif\">");
     out.println("<h2><font color=\"black\">");
     out.println(getHelloSEIPort().sayHello(username));
     out.println("</font></h2>");
  } catch(java.rmi.RemoteException ex) {
  ex.printStackTrace(out);
  }
} else {
  out.println("You didn't specify your name.<br/>");
}
out.println("<a href=\"index.jsp\">back</a>");
out.println("</body>");
```

7. Expand the project's Web Pages node, double-click the default index.jsp file, and replace the <body> tags with the following code:

```
     <body bgcolor="white">
       <img src="duke.waving.gif">
       <h2>Hello, my name is Duke. What's yours?</h2>
       <form method="get" action="HelloServlet">
          <input type="text" name="username" size="25">
          <br/>
          <input type="submit" value="Submit">
          <input type="reset" value="Reset">
       </form>
     </body>
```

8. Go to <*INSTALL*>`/j2eetutorial14/examples/jaxrpc/webclient/web` and copy the `duke.waving.gif` file into your project's `web` directory.

Building and Deploying the Static Stub Client

In the IDE, perform these steps:

1. In the Projects window, right-click the project node.
2. Choose Run Project.

This task invokes the web service client. When you run this task, the browser opens, the application is displayed, and you can submit a name, and a greeting is returned.

IDE-Generated Static Stub Client

To build, package, and run the client, follow these steps:

1. Choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to <*INSTALL*>`/j2eetutorial14/examples/jaxrpc/staticstub/`, select the project, and choose Open Project.
2. The project prompts you to set up a library named "jax-rpc". The library should contain JAR files that are needed by the project. Right-click the project and choose Resolve Reference Problems. Click Resolve. Click New Library and name the library `jax-rpc`. Click Add JAR/Folder and

navigate to the `lib` directory in your application server installation. Select `activation.jar, dom.jar, j2ee.jar, jaxrpc-api.jar, jaxrpc-impl.jar, jhall.jar, mail.jar, saaj-impl.jar, xercesImpl.jar` and click OK.

3. In the Projects window, right-click the project and choose Run Project. The IDE builds, packages, and runs the project.
4. In the Output window, the client displays the following output:

```
Hello Duke!
```

Before it can invoke the remote methods on the stub, the client performs these steps:

1. Creates a `Stub` object:

   ```
   (Stub)(new Hello_Impl().getHelloSEIport())
   ```

   The code in this method is implementation-specific because it relies on a `Hello_Impl` object, which is not defined in the specifications. The `Hello_Impl` class will be generated by `wscompile` in the following section.

2. Sets the endpoint address that the stub uses to access the service:

   ```
   stub._setProperty
   (javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY, args[0]);
   ```

   At runtime, the endpoint address is passed to `HelloClient` in `args[0]` as a command-line parameter, which the IDE gets from the `endpoint.address` property in the `build.properties` file. This address must match the one you set for the service in Specifying the Endpoint Address (page 35).

3. Casts `stub` to the service endpoint interface, `HelloSEI`:

   ```
   HelloSEI hello = (HelloSEI)stub;
   ```

Here is the full source code listing for the `HelloClient.java` file, which is located in the directory *<INSTALL>*`/j2eetutorial14/examples/jaxrpc/staticstub/src/`:

```
package staticstub;

import javax.xml.rpc.Stub;

public class HelloClient {

    private String endpointAddress;
```

```
        public static void main(String[] args) {

            System.out.println("Endpoint address = " + args[0]);
            try {
                Stub stub = createProxy();
                stub._setProperty
                  (javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,
                   args[0]);
                HelloSEI hello = (HelloSEI)stub;
                System.out.println(hello.sayHello("Duke!"));
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }

        private static Stub createProxy() {
            // Note: Hello_Impl is implementation-specific.
            return
            (Stub) (new Hello_Impl().getHelloSEIPort());
        }
    }
```

Building the Static Stub Client

To build the client, right-click its node in the Projects window and choose Build Project. The Build Project command executes several subtasks in your Ant build script, the most important of which are the following:

Running the Static Stub Client

In the IDE, perform these steps:

1. In the Projects window, right-click the project.
2. Choose Run Project.

This task invokes the web service client. When you run this task, you should get the following output:

```
Hello Duke!
```

Types Supported by JAX-RPC

Behind the scenes, JAX-RPC maps types of the Java programming language to XML/WSDL definitions. For example, JAX-RPC maps the `java.lang.String` class to the `xsd:string` XML data type. Application developers don't need to know the details of these mappings, but they should be aware that not every class in the Java 2 Platform, Standard Edition (J2SE) can be used as a method parameter or return type in JAX-RPC.

J2SE SDK Classes

JAX-RPC supports the following J2SE SDK classes:

```
java.lang.Boolean
java.lang.Byte
java.lang.Double
java.lang.Float
java.lang.Integer
java.lang.Long
java.lang.Short
java.lang.String

java.math.BigDecimal
java.math.BigInteger

java.net.URI

java.util.Calendar
java.util.Date
```

Primitives

JAX-RPC supports the following primitive types of the Java programming language:

```
boolean
byte
double
float
int
long
short
```

Arrays

JAX-RPC also supports arrays that have members of supported JAX-RPC types. Examples of supported arrays are `int[]` and `String[]`. Multidimensional arrays, such as `BigDecimal[][]`, are also supported.

Value Types

A *value type* is a class whose state can be passed between a client and a remote service as a method parameter or return value. For example, in an application for a university library, a client might call a remote procedure with a value type parameter named `Book`, a class that contains the fields `Title`, `Author`, and `Publisher`.

To be supported by JAX-RPC, a value type must conform to the following rules:

- It must have a public default constructor.
- It must not implement (either directly or indirectly) the `java.rmi.Remote` interface.
- Its fields must be supported JAX-RPC types.

The value type can contain public, private, or protected fields. The field of a value type must meet these requirements:

- A public field cannot be final or transient.
- A nonpublic field must have corresponding getter and setter methods.

Web Service Clients

This section shows how to create and run these types of clients:

- Dynamic proxy
- Dynamic invocation interface (DII)

When you run these client examples, they will access the `MyHelloService` that you deployed in Creating a Simple Web Service and Client with JAX-RPC (page 30).

Dynamic Proxy Client

The client in the preceding section uses a static stub for the proxy. In contrast, the client example in this section calls a remote procedure through a *dynamic proxy*, a class that is created during runtime. Although the source code for the IDE-generated static stub client relies on an implementation-specific class, the code for the dynamic proxy client does not have this limitation, just like the J2EE container-generated static stub.

This example resides in the *<INSTALL>*`/j2eetutorial14/examples/jaxrpc/dynamicproxy/` directory.

Coding the Dynamic Proxy Client

The `HelloDProxyClient` program constructs the dynamic proxy as follows:

1. Creates a `Service` object named `helloService`:

   ```
   Service helloService =
       serviceFactory.createService(helloWsdlUrl,
       new QName(nameSpaceUri, serviceName));
   ```

   A `Service` object is a factory for proxies. To create the `Service` object (`helloService`), the program calls the `createService` method on another type of factory, a `ServiceFactory` object.

   The `createService` method has two parameters: the URL of the WSDL file and a `QName` object. At runtime, the client gets information about the service by looking up its WSDL. In this example, the URL of the WSDL file points to the WSDL that was deployed with `HelloService`:

   ```
   http://localhost:8080/helloservice/Hello?WSDL
   ```

   A `QName` object is a tuple that represents an XML qualified name. The tuple is composed of a namespace URI and the local part of the qualified name. In the `QName` parameter of the `createService` invocation, the local part is the service name, `HelloService`.

2. The program creates a proxy (`myProxy`) with a type of the service endpoint interface (`HelloSEI`):

   ```
   dynamicproxy.HelloSEI myProxy =
      (dynamicproxy.HelloSEI)helloService.getPort(
      new QName(nameSpaceUri, portName),
      dynamicproxy.HelloSEI.class);
   ```

The `helloService` object is a factory for dynamic proxies. To create `myProxy`, the program calls the `getPort` method of `helloService`. This method has two parameters: a `QName` object that specifies the port name and a `java.lang.Class` object for the service endpoint interface (`HelloSEI`). The `HelloSEI` class is generated by `wscompile`. The port name (`HelloSEIPort`) is specified by the WSDL file.

Here is the listing for the `HelloDProxyClient.java` file, located in the *<INSTALL>*`/j2eetutorial14/examples/jaxrpc/dynamicproxy/src/dynamicproxy` directory:

```
package dynamicproxy;

import java.net.URL;
import javax.xml.rpc.Service;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceFactory;
import dynamicproxy.HelloIF;

public class HelloDProxyClient {

    public static void main(String[] args) {
        try {

            String UrlString = args[0] + "?WSDL";
            String nameSpaceUri = "urn:Hello/wsdl";
            String serviceName = "Hello";
            String portName = "HelloSEIPort";

            System.out.println("UrlString = " + UrlString);
            URL helloWsdlUrl = new URL(UrlString);

            ServiceFactory serviceFactory =
                ServiceFactory.newInstance();

            Service helloService =
                serviceFactory.createService(helloWsdlUrl,
                new QName(nameSpaceUri, serviceName));

            dynamicproxy.HelloSEI myProxy =
                (dynamicproxy.HelloSEI)
                helloService.getPort(
                new QName(nameSpaceUri, portName),
                dynamicproxy.HelloSEI.class);

            System.out.println(myProxy.sayHello("Buzz"));
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Building and Running the Dynamic Proxy Client

Before performing the steps in this section, you must first create and deploy `HelloService` as described in Creating a Simple Web Service and Client with JAX-RPC (page 30).

To build, package, and run the client, follow these steps:

1. If you have not already opened the dynamicproxy project, choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to `<`*INSTALL*`>/j2eetutorial14/examples/jaxrpc/dynamicproxy/`, select the project, and choose Open Project Folder.
2. If you have not already created the JAX-RPC library, the project prompts you to set it up. The library should contain JAR files that are needed by the project. Right-click the project and choose Resolve Reference Problems. Click Resolve. Click New Library and name the library `jax-rpc`. Click Add JAR/Folder and navigate to the `lib` directory in your application server installation. Select `activation.jar`, `dom.jar`, `j2ee.jar`, `jaxrpc-api.jar`, `jaxrpc-impl.jar`, `jhall.jar`, `mail.jar`, `saaj-impl.jar`, `xercesImpl.jar` and click OK. Click Close.
3. In the Projects window, right-click the project and choose Run Project. The IDE builds, packages, and runs the project.
4. In the Output window, the client displays the following output:

```
Hello Buzz
```

Dynamic Invocation Interface Client

With the dynamic invocation interface (DII), a client can call a remote procedure even if the signature of the remote procedure or the name of the service is unknown until runtime. In contrast to a static stub or dynamic proxy client, a DII client does not require runtime classes generated by `wscompile`. However, as you'll see in the following section, the source code for a DII client is more complicated than the code for the other two types of clients.

This example is for advanced users who are familiar with WSDL documents. (See Further Information, page 51.)

This example resides in the *<INSTALL>*`/j2eetutorial14/examples/jaxrpc/diiclient/` directory.

Coding the DII Client

The `HelloDIIClient` program performs these steps:

1. Creates a `Service` object:

   ```
   Service service =
   ```

5. Specifies the method's return type, name, and parameter:

   ```
   QName QNAME_TYPE_STRING = new QName(NS_XSD, "string");
   call.setReturnType(QNAME_TYPE_STRING);

   call.setOperationName(new QName(BODY_NAMESPACE_VALUE,
       "sayHello"));

   call.addParameter("String_1", QNAME_TYPE_STRING,
       ParameterMode.IN);
   ```

   To specify the return type, the program invokes the `setReturnType` method on the `Call` object. The parameter of `setReturnType` is a `QName` object that represents an XML string type.

   The program designates the method name by invoking the `setOperationName` method with a `QName` object that represents `sayHello`.

   To indicate the method parameter, the program invokes the `addParameter` method on the `Call` object. The `addParameter` method has three arguments: a `String` for the parameter name (`String_1`), a `QName` object for the XML type, and a `ParameterMode` object to indicate the passing mode of the parameter (`IN`).

6. Invokes the remote method on the `Call` object:

   ```
   String[] params = { "Murphy" };
   String result = (String)call.invoke(params);
   ```

   The program assigns the parameter value (`Murphy`) to a `String` array (`params`) and then executes the `invoke` method with the `String` array as an argument.

Here is the listing for the `HelloClient.java` file, located in the *<INSTALL>*`/j2eetutorial14/examples/jaxrpc/dii/src/` directory:

```
package diiclient;

import javax.xml.rpc.Call;
import javax.xml.rpc.Service;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.ParameterMode;

public class HelloDIIClient {

    private static String qnameService = "Hello";
    private static String qnamePort = "HelloSEI";
```

```
private static String BODY_NAMESPACE_VALUE =
    "urn:Hello/wsdl";
private static String ENCODING_STYLE_PROPERTY =
     "javax.xml.rpc.encodingstyle.namespace.uri";
private static String NS_XSD =
    "http://www.w3.org/2001/XMLSchema";
private static String URI_ENCODING =
     "http://schemas.xmlsoap.org/soap/encoding/";

public static void main(String[] args) {

    System.out.println("Endpoint address = " + args[0]);

    try {
        ServiceFactory factory =
            ServiceFactory.newInstance();
        Service service =
            factory.createService(
            new QName(qnameService));

        QName port = new QName(qnamePort);

        Call call = service.createCall(port);
        call.setTargetEndpointAddress(args[0]);

        call.setProperty(Call.SOAPACTION_USE_PROPERTY,
            new Boolean(true));
        call.setProperty(Call.SOAPACTION_URI_PROPERTY
             "");
        call.setProperty(ENCODING_STYLE_PROPERTY,
            URI_ENCODING);
        QName QNAME_TYPE_STRING =
                    new QName(NS_XSD, "string");
        call.setReturnType(QNAME_TYPE_STRING);

        call.setOperationName(
            new QName(BODY_NAMESPACE_VALUE,"sayHello"));
        call.addParameter("String_1", QNAME_TYPE_STRING,
            ParameterMode.IN);
        String[] params = { "Murph!" };

        String result = (String)call.invoke(params);
        System.out.println(result);

    } catch (Exception ex) {
```

```
                ex.printStackTrace();
            }
        }
    }
```

Building and Running the DII Client

Before performing the steps in this section, you must first create and deploy `HelloService` as described in Creating a Simple Web Service and Client with JAX-RPC (page 30).

To build, package, and run the client, follow these steps:

1. If you have not already opened the DIIClient project, choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to <*INSTALL*>`/j2eetutorial14/examples/jaxrpc/diiclient/`, select the project, and choose Open Project.
2. If you have not already created the JAX-RPC library, the project prompts you to set it up. The library should contain JAR files that are needed by the project. Right-click the project and choose Resolve Reference Problems. Click Resolve. Click New Library and name the library `jax-rpc.` Click Add JAR/Folder and navigate to the `lib` directory in your application server installation. Select `activation.jar, dom.jar, j2ee.jar, jaxrpc-api.jar, jaxrpc-impl.jar, jhall.jar, mail.jar, saaj-impl.jar, xercesImpl.jar` and click OK. Click Close.
3. In the Projects window, right-click the project and choose Run Project. The IDE builds, packages, and runs the project.
4. In the Output window, the client displays the following output:

```
Hello Murph!
```

Web Services Interoperability and JAX-RPC

JAX-RPC 1.1 supports the Web Services Interoperability (WS-I) Basic Profile Version 1.0, Working Group Approval Draft. The WS-I Basic Profile is a document that clarifies the SOAP 1.1 and WSDL 1.1 specifications in order to promote SOAP interoperability. For links related to WS-I, see Further Information (page 51).

To support WS-I, JAX-RPC has the following features:

- When run with the `-f:wsi` option, `wscompile` verifies that a WSDL is WS-I-compliant or generates classes needed by JAX-RPC services and clients that are WS-I-compliant.
- The JAX-RPC runtime supports doc/literal and rpc/literal encodings for services, static stubs, dynamic proxies, and DII.

You can set these properties by right-clicking a project, choosing Properties, and clicking Web Services or web Service Clients.

Further Information

For more information about JAX-RPC and related technologies, refer to the following:

- Java API for XML-based RPC 1.1 specification
  `http://java.sun.com/xml/downloads/jaxrpc.html`
- JAX-RPC home
  `http://java.sun.com/xml/jaxrpc/`
- Simple Object Access Protocol (SOAP) 1.1 W3C Note
  `http://www.w3.org/TR/SOAP/`
- Web Services Description Language (WSDL) 1.1 W3C Note
  `http://www.w3.org/TR/wsdl`
- WS-I Basic Profile 1.0
  `http://www.ws-i.org`

3

SOAP with Attachments API for Java

SOAP with Attachments API for Java (SAAJ) is used mainly for the SOAP messaging that goes on behind the scenes in JAX-RPC and JAXR implementations. Secondarily, it is an API that developers can use when they choose to write SOAP messaging applications directly rather than use JAX-RPC. The SAAJ API allows you to do XML messaging from the Java platform: By simply making method calls using the SAAJ API, you can read and write SOAP-based XML messages, and you can optionally send and receive such messages over the Internet (some implementations may not support sending and receiving). This chapter will help you learn how to use the SAAJ API.

The SAAJ API conforms to the Simple Object Access Protocol (SOAP) 1.1 specification and the SOAP with Attachments specification. The SAAJ 1.2 specification defines the javax.xml.soap package, which contains the API for creating and populating a SOAP message. This package has all the API necessary for sending request-response messages. (Request-response messages are explained in SOAPConnection Objects, page 58.)

Note: The javax.xml.messaging package, defined in the Java API for XML Messaging (JAXM) 1.1 specification, is not part of the J2EE 1.4 platform and is not discussed in this chapter. The JAXM API is available as a separate download from http://java.sun.com/xml/jaxm/.

This chapter starts with an overview of messages and connections, giving some of the conceptual background behind the SAAJ API to help you understand why certain things are done the way they are. Next, the tutorial shows you how to use the basic SAAJ API, giving examples and explanations of the commonly used features. The code examples in the last part of the tutorial show you how to build an application.

Overview of SAAJ

This section presents a high-level view of how SAAJ messaging works and explains concepts in general terms. Its goal is to give you some terminology and a framework for the explanations and code examples that are presented in the tutorial section.

The overview looks at SAAJ from two perspectives: messages and connections.

Messages

SAAJ messages follow SOAP standards, which prescribe the format for messages and also specify some things that are required, optional, or not allowed. With the SAAJ API, you can create XML messages that conform to the SOAP 1.1 and WS-I Basic Profile 1.0 specifications simply by making Java API calls.

The Structure of an XML Document

An XML document has a hierarchical structure made up of elements, subelements, subsubelements, and so on. You will notice that many of the SAAJ classes and interfaces represent XML elements in a SOAP message and have the word *element* or *SOAP* (or both) in their names.

An element is also referred to as a *node*. Accordingly, the SAAJ API has the interface Node, which is the base class for all the classes and interfaces that represent XML elements in a SOAP message. There are also methods such as SOAPElement.addTextNode, Node.detachNode, and Node.getValue, which you will see how to use in the tutorial section.

What Is in a Message?

The two main types of SOAP messages are those that have attachments and those that do not.

Messages with No Attachments

The following outline shows the very high-level structure of a SOAP message with no attachments. Except for the SOAP header, all the parts listed are required to be in every SOAP message.

I. SOAP message

  A. SOAP part

    1. SOAP envelope

      a. SOAP header (optional)

      b. SOAP body

The SAAJ API provides the SOAPMessage class to represent a SOAP message, the SOAPPart class to represent the SOAP part, the SOAPEnvelope interface to represent the SOAP envelope, and so on. Figure 3–1 illustrates the structure of a SOAP message with no attachments.

Note: Many SAAJ API interfaces extend DOM interfaces. In a SAAJ message, the SOAPPart class is also a DOM document. See SAAJ and DOM (page 58) for details.

When you create a new SOAPMessage object, it will automatically have the parts that are required to be in a SOAP message. In other words, a new SOAPMessage object has a SOAPPart object that contains a SOAPEnvelope object. The SOAPEnvelope object in turn automatically contains an empty SOAPHeader object followed by an empty SOAPBody object. If you do not need the SOAPHeader object, which is optional, you can delete it. The rationale for having it automatically included is that more often than not you will need it, so it is more convenient to have it provided.

The SOAPHeader object can include one or more headers that contain metadata about the message (for example, information about the sending and receiving parties). The SOAPBody object, which always follows the SOAPHeader object if there is one, contains the message content. If there is a SOAPFault object (see Using SOAP Faults, page 80), it must be in the SOAPBody object.

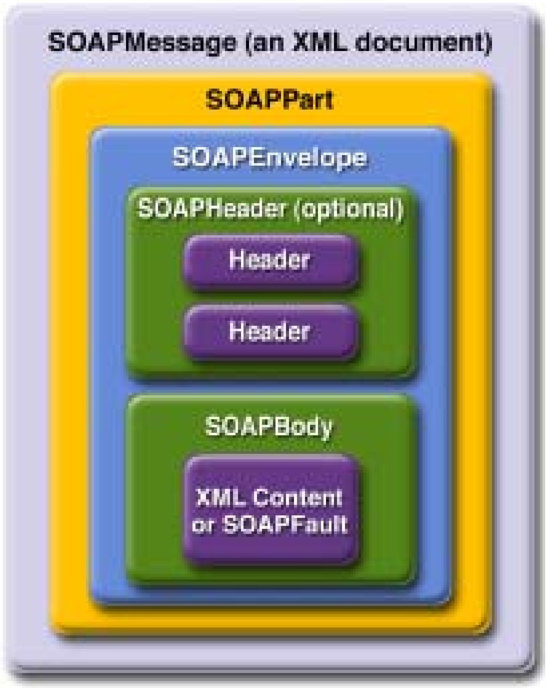


Figure 3–1 SOAPMessage Object with No Attachments

Messages with Attachments

A SOAP message may include one or more attachment parts in addition to the SOAP part. The SOAP part must contain only XML content; as a result, if any of the content of a message is not in XML format, it must occur in an attachment part. So if, for example, you want your message to contain a binary file, your message must have an attachment part for it. Note that an attachment part can contain any kind of content, so it can contain data in XML format as well. Figure 3–2 shows the high-level structure of a SOAP message that has two attachments.

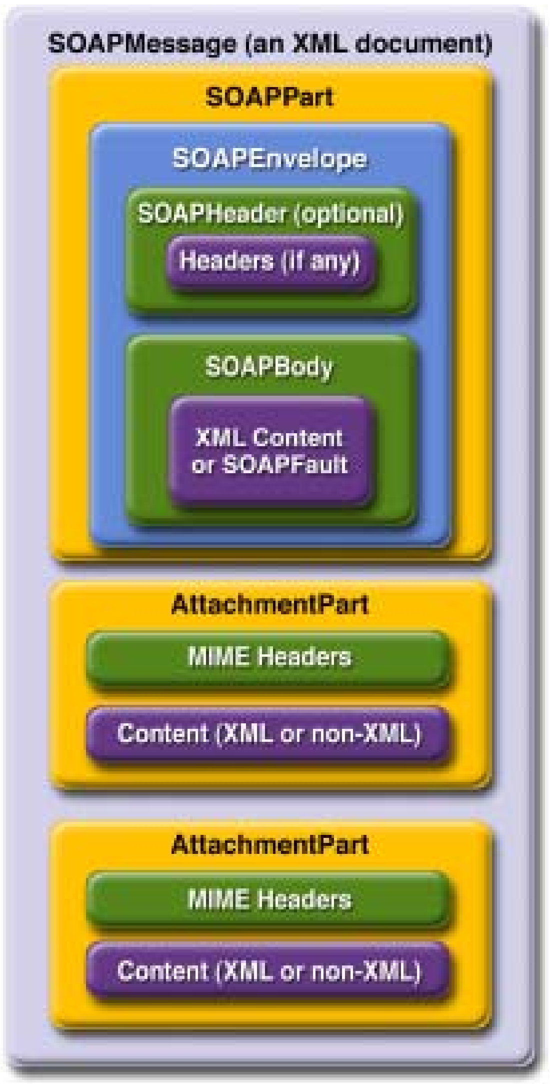


Figure 3–2 SOAPMessage Object with Two AttachmentPart Objects

The SAAJ API provides the AttachmentPart class to represent an attachment part of a SOAP message. A SOAPMessage object automatically has a SOAPPart object and its required subelements, but because AttachmentPart objects are optional, you must create and add them yourself. The tutorial section walks you through creating and populating messages with and without attachment parts.

If a SOAPMessage object has one or more attachments, each AttachmentPart object must have a MIME header to indicate the type of data it contains. It may also have additional MIME headers to identify it or to give its location. These headers are optional but can be useful when there are multiple attachments. When a SOAPMessage object has one or more AttachmentPart objects, its SOAPPart object may or may not contain message content.

SAAJ and DOM

In SAAJ 1.2, the SAAJ APIs extend their counterparts in the org.w3c.dom package:

- The Node interface extends the org.w3c.dom.Node interface.
- The SOAPElement interface extends both the Node interface and the org.w3c.dom.Element interface.
- The SOAPPart class implements the org.w3c.dom.Document interface.
- The Text interface extends the org.w3c.dom.Text interface.

Moreover, the SOAPPart of a SOAPMessage is also a DOM Level 2 Document and can be manipulated as such by applications, tools, and libraries that use DOM. For details on how to use DOM documents with the SAAJ API, see Adding Content to the SOAPPart Object (page 70) and Adding a Document to the SOAP Body (page 71).

Connections

All SOAP messages are sent and received over a connection. With the SAAJ API, the connection is represented by a SOAPConnection object, which goes from the sender directly to its destination. This kind of connection is called a *point-to-point* connection because it goes from one endpoint to another endpoint. Messages sent using the SAAJ API are called *request-response messages*. They are sent over a SOAPConnection object with the call method, which sends a message (a request) and then blocks until it receives the reply (a response).

SOAPConnection Objects

The following code fragment creates the SOAPConnection object connection and then, after creating and populating the message, uses connection to send the message. As stated previously, all messages sent over a SOAPConnection object are sent with the call method, which both sends the message and blocks until it receives the response. Thus, the return value for the call method is the SOAPMessage object that is the response to the message that was sent. The request parameter is the message being sent; endpoint represents where it is being sent.

```
SOAPConnectionFactory factory =
    SOAPConnectionFactory.newInstance();
SOAPConnection connection = factory.createConnection();

. . .// create a request message and give it content

java.net.URL endpoint =
    new URL("http://fabulous.com/gizmo/order");
SOAPMessage response = connection.call(request, endpoint);
```

Note that the second argument to the call method, which identifies where the message is being sent, can be a String object or a URL object. Thus, the last two lines of code from the preceding example could also have been the following:

```
String endpoint = "http://fabulous.com/gizmo/order";
SOAPMessage response = connection.call(request, endpoint);
```

A web service implemented for request-response messaging must return a response to any message it receives. The response is a SOAPMessage object, just as the request is a SOAPMessage object. When the request message is an update, the response is an acknowledgment that the update was received. Such an acknowledgment implies that the update was successful. Some messages may not require any response at all. The service that gets such a message is still required to send back a response because one is needed to unblock the call method. In this case, the response is not related to the content of the message; it is simply a message to unblock the call method.

Now that you have some background on SOAP messages and SOAP connections, in the next section you will see how to use the SAAJ API.

Tutorial

This tutorial walks you through how to use the SAAJ API. First, it covers the basics of creating and sending a simple SOAP message. Then you will learn more details about adding content to messages, including how to create SOAP faults and attributes. Finally, you will learn how to send a message and retrieve the content of the response. After going through this tutorial, you will know how to perform the following tasks:

- Creating and sending a simple message
- Adding content to the header
- Adding content to the SOAPPart object
- Adding a document to the SOAP body
- Manipulating message content using SAAJ or DOM APIs
- Adding attachments
- Adding attributes
- Using SOAP faults

In the section Code Examples (page 85), you will see the code fragments from earlier parts of the tutorial in runnable applications, which you can test yourself.

A SAAJ client can send request-response messages to web services that are implemented to do request-response messaging. This section demonstrates how you can do this.

Creating and Sending a Simple Message

This section covers the basics of creating and sending a simple message and retrieving the content of the response. It includes the following topics:

- Creating a message
- Parts of a message
- Accessing elements of a message
- Adding content to the body
- Getting a SOAPConnection object
- Sending a message
- Getting the content of a message

Creating a Message

The first step is to create a message using a MessageFactory object. The SAAJ API provides a default implementation of the MessageFactory class, thus making it easy to get an instance. The following code fragment illustrates getting an instance of the default message factory and then using it to create a message.

```
MessageFactory factory = MessageFactory.newInstance();
SOAPMessage message = factory.createMessage();
```

As is true of the newInstance method for SOAPConnectionFactory, the newInstance method for MessageFactory is static, so you invoke it by calling MessageFactory.newInstance.

Parts of a Message

A SOAPMessage object is required to have certain elements, and, as stated previously, the SAAJ API simplifies things for you by returning a new SOAPMessage object that already contains these elements. So message, which was created in the preceding line of code, automatically has the following:

I. A SOAPPart object that contains
    A. A SOAPEnvelope object that contains
        1. An empty SOAPHeader object
        2. An empty SOAPBody object

The SOAPHeader object is optional and can be deleted if it is not needed. However, if there is one, it must precede the SOAPBody object. The SOAPBody object can hold either the content of the message or a *fault* message that contains status information or details about a problem with the message. The section Using SOAP Faults (page 80) walks you through how to use SOAPFault objects.

Accessing Elements of a Message

The next step in creating a message is to access its parts so that content can be added. There are two ways to do this. The SOAPMessage object message, created in the preceding code fragment, is the place to start.

The first way to access the parts of the message is to work your way through the structure of the message. The message contains a SOAPPart object, so you use the getSOAPPart method of message to retrieve it:

```
SOAPPart soapPart = message.getSOAPPart();
```

Next you can use the getEnvelope method of soapPart to retrieve the SOAPEnvelope object that it contains.

```
SOAPEnvelope envelope = soapPart.getEnvelope();
```

You can now use the getHeader and getBody methods of envelope to retrieve its empty SOAPHeader and SOAPBody objects.

```
SOAPHeader header = envelope.getHeader();
SOAPBody body = envelope.getBody();
```

The second way to access the parts of the message is to retrieve the message header and body directly, without retrieving the SOAPPart or SOAPEnvelope. To do so, use the getSOAPHeader and getSOAPBody methods of SOAPMessage:

```
SOAPHeader header = message.getSOAPHeader();
SOAPBody body = message.getSOAPBody();
```

This example of a SAAJ client does not use a SOAP header, so you can delete it. (You will see more about headers later.) Because all SOAPElement objects, including SOAPHeader objects, are derived from the Node interface, you use the method Node.detachNode to delete header.

```
header.detachNode();
```

Adding Content to the Body

The SOAPBody object contains either content or a fault. To add content to the body, you normally create one or more SOAPBodyElement objects to hold the content. You can also add subelements to the SOAPBodyElement objects by using the addChildElement method. For each element or child element, you add content by using the addTextNode method.

When you create any new element, you also need to create an associated Name object so that it is uniquely identified. One way to create Name objects is by using SOAPEnvelope methods, so you can use the envelope variable from the earlier code fragment to create the Name object for your new element. Another way to create Name objects is to use SOAPFactory methods, which are useful if you do not have access to the SOAPEnvelope.

Note: The SOAPFactory class also lets you create XML elements when you are not creating an entire message or do not have access to a complete SOAPMessage object.

For example, JAX-RPC implementations often work with XML fragments rather than complete SOAPMessage objects. Consequently, they do not have access to a SOAPEnvelope object, and this makes using a SOAPFactory object to create Name objects very useful. In addition to a method for creating Name objects, the SOAPFactory class provides methods for creating Detail objects and SOAP fragments. You will find an explanation of Detail objects in Overview of SOAP Faults (page 80) and Creating and Populating a SOAPFault Object (page 82).

Name objects associated with SOAPBodyElement or SOAPHeaderElement objects must be fully qualified; that is, they must be created with a local name, a prefix for the namespace being used, and a URI for the namespace. Specifying a namespace for an element makes clear which one is meant if more than one element has the same local name.

The following code fragment retrieves the SOAPBody object body from message, uses a SOAPFactory to create a Name object for the element to be added, and adds a new SOAPBodyElement object to body.

```
SOAPBody body = message.getSOAPBody();
SOAPFactory soapFactory = SOAPFactory.newInstance();
Name bodyName = soapFactory.createName("GetLastTradePrice",
    "m", "http://wombat.ztrade.com");
SOAPBodyElement bodyElement = body.addBodyElement(bodyName);
```

At this point, body contains a SOAPBodyElement object identified by the Name object bodyName, but there is still no content in bodyElement. Assuming that you want to get a quote for the stock of Sun Microsystems, Inc., you need to create a child element for the symbol using the addChildElement method. Then you need to give it the stock symbol using the addTextNode method. The Name object for the new SOAPElement object symbol is initialized with only a local name because child elements inherit the prefix and URI from the parent element.

```
Name name = soapFactory.createName("symbol");
SOAPElement symbol = bodyElement.addChildElement(name);
symbol.addTextNode("SUNW");
```

You might recall that the headers and content in a SOAPPart object must be in XML format. The SAAJ API takes care of this for you, building the appropriate XML constructs automatically when you call methods such as addBodyElement, addChildElement, and addTextNode. Note that you can call the method addTextNode only on an element such as bodyElement or any child elements that are added to it. You cannot call addTextNode on a SOAPHeader or SOAPBody object because they contain elements and not text.

The content that you have just added to your SOAPBody object will look like the following when it is sent over the wire:

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <m:GetLastTradePrice xmlns:m="http://wombat.ztrade.com">
      <symbol>SUNW</symbol>
    </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Let’s examine this XML excerpt line by line to see how it relates to your SAAJ code. Note that an XML parser does not care about indentations, but they are generally used to indicate element levels and thereby make it easier for a human reader to understand.

Here is the SAAJ code:

```
SOAPMessage message = messageFactory.createMessage();
SOAPHeader header = message.getSOAPHeader();
SOAPBody body = message.getSOAPBody();
```

Here is the XML it produces:

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body>
    . . .
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The outermost element in this XML example is the SOAP envelope element, indicated by SOAP-ENV:Envelope. Note that Envelope is the name of the element, and SOAP-ENV is the namespace prefix. The interface SOAPEnvelope represents a SOAP envelope.

The first line signals the beginning of the SOAP envelope element, and the last line signals the end of it; everything in between is part of the SOAP envelope. The second line is an example of an attribute for the SOAP envelope element. Because a SOAP envelope element always contains this attribute with this value, a SOAPMessage object comes with it automatically included. xmlns stands for “XML namespace,” and its value is the URI of the namespace associated with Envelope.

The next line is an empty SOAP header. We could remove it by calling header.detachNode after the getSOAPHeader call.

The next two lines mark the beginning and end of the SOAP body, represented in SAAJ by a SOAPBody object. The next step is to add content to the body.

Here is the SAAJ code:

```
Name bodyName = soapFactory.createName("GetLastTradePrice",
    "m", "http://wombat.ztrade.com");
SOAPBodyElement bodyElement = body.addBodyElement(bodyName);
```

Here is the XML it produces:

```
<m:GetLastTradePrice
 xmlns:m="http://wombat.ztrade.com">
 . . . .
</m:GetLastTradePrice>
```

These lines are what the SOAPBodyElement bodyElement in your code represents. GetLastTradePrice is its local name, m is its namespace prefix, and http://wombat.ztrade.com is its namespace URI.

Here is the SAAJ code:

```
Name name = soapFactory.createName("symbol");
SOAPElement symbol = bodyElement.addChildElement(name);
symbol.addTextNode("SUNW");
```

Here is the XML it produces:

```
<symbol>SUNW</symbol>
```

The String "SUNW" is the text node for the element <symbol>. This String object is the message content that your recipient, the stock quote service, receives.

The following example shows how to add multiple SOAPElement objects and add text to each of them. The code first creates the SOAPBodyElement object purchaseLineItems, which has a fully qualified name associated with it. That is, the Name object for it has a local name, a namespace prefix, and a namespace URI. As you saw earlier, a SOAPBodyElement object is required to have a fully qualified name, but child elements added to it, such as SOAPElement objects, can have Name objects with only the local name.

```
SOAPBody body = message.getSOAPBody();
Name bodyName = soapFactory.createName("PurchaseLineItems",
    "PO", "http://sonata.fruitsgalore.com");
SOAPBodyElement purchaseLineItems =
    body.addBodyElement(bodyName);

Name childName = soapFactory.createName("Order");
SOAPElement order =
    purchaseLineItems.addChildElement(childName);

childName = soapFactory.createName("Product");
SOAPElement product = order.addChildElement(childName);
product.addTextNode("Apple");

childName = soapFactory.createName("Price");
SOAPElement price = order.addChildElement(childName);
price.addTextNode("1.56");

childName = soapFactory.createName("Order");
SOAPElement order2 =
    purchaseLineItems.addChildElement(childName);

childName = soapFactory.createName("Product");
SOAPElement product2 = order2.addChildElement(childName);
product2.addTextNode("Peach");

childName = soapFactory.createName("Price");
SOAPElement price2 = order2.addChildElement(childName);
price2.addTextNode("1.48");
```

The SAAJ code in the preceding example produces the following XML in the SOAP body:

```
<PO:PurchaseLineItems
 xmlns:PO="http://sonata.fruitsgalore.com">
  <Order>
    <Product>Apple</Product>
    <Price>1.56</Price>
  </Order>

  <Order>
```

```
    <Product>Peach</Product>
    <Price>1.48</Price>
  </Order>
</PO:PurchaseLineItems>
```

Getting a SOAPConnection Object

The SAAJ API is focused primarily on reading and writing messages. After you have written a message, you can send it using various mechanisms (such as JMS or JAXM). The SAAJ API does, however, provide a simple mechanism for request-response messaging.

To send a message, a SAAJ client can use a SOAPConnection object. A SOAPConnection object is a point-to-point connection, meaning that it goes directly from the sender to the destination (usually a URL) that the sender specifies.

The first step is to obtain a SOAPConnectionFactory object that you can use to create your connection. The SAAJ API makes this easy by providing the SOAPConnectionFactory class with a default implementation. You can get an instance of this implementation using the following line of code.

```
SOAPConnectionFactory soapConnectionFactory =
    SOAPConnectionFactory.newInstance();
```

Now you can use soapConnectionFactory to create a SOAPConnection object.

```
SOAPConnection connection =
    soapConnectionFactory.createConnection();
```

You will use connection to send the message that you created.

Sending a Message

A SAAJ client calls the SOAPConnection method call on a SOAPConnection object to send a message. The call method takes two arguments: the message being sent and the destination to which the message should go. This message is going to the stock quote service indicated by the URL object endpoint.

```
java.net.URL endpoint = new URL(
    "http://wombat.ztrade.com/quotes");

SOAPMessage response = connection.call(message, endpoint);
```

The content of the message you sent is the stock symbol SUNW; the SOAPMessage object response should contain the last stock price for Sun Microsystems, which you will retrieve in the next section.

A connection uses a fair amount of resources, so it is a good idea to close a connection as soon as you are finished using it.

```
connection.close();
```

Getting the Content of a Message

The initial steps for retrieving a message's content are the same as those for giving content to a message: Either you use the Message object to get the SOAPBody object, or you access the SOAPBody object through the SOAPPart and SOAPEnvelope objects.

Then you access the SOAPBody object's SOAPBodyElement object, because that is the element to which content was added in the example. (In a later section you will see how to add content directly to the SOAPPart object, in which case you would not need to access the SOAPBodyElement object to add content or to retrieve it.)

To get the content, which was added with the method SOAPElement.addTextNode, you call the method Node.getValue. Note that getValue returns the value of the immediate child of the element that calls the method. Therefore, in the following code fragment, the getValue method is called on bodyElement, the element on which the addTextNode method was called.

To access bodyElement, you call the getChildElements method on soapBody. Passing bodyName to getChildElements returns a java.util.Iterator object that contains all the child elements identified by the Name object bodyName. You already know that there is only one, so calling the next method on it will return the SOAPBodyElement you want. Note that the Iterator.next method returns a Java Object, so you need to cast the Object it returns to a SOAPBodyElement object before assigning it to the variable bodyElement.

```
SOAPBody soapBody = response.getSOAPBody();
java.util.Iterator iterator =
    soapBody.getChildElements(bodyName);
SOAPBodyElement bodyElement =
    (SOAPBodyElement)iterator.next();
String lastPrice = bodyElement.getValue();
System.out.print("The last price for SUNW is ");
System.out.println(lastPrice);
```

If more than one element had the name bodyName, you would have to use a while loop using the Iterator.hasNext method to make sure that you got all of them.

```
while (iterator.hasNext()) {
    SOAPBodyElement bodyElement =
        (SOAPBodyElement)iterator.next();
    String lastPrice = bodyElement.getValue();
    System.out.print("The last price for SUNW is ");
    System.out.println(lastPrice);
}
```

At this point, you have seen how to send a very basic request-response message and get the content from the response. The next sections provide more detail on adding content to messages.

Adding Content to the Header

To add content to the header, you create a SOAPHeaderElement object. As with all new elements, it must have an associated Name object, which you can create using the message's SOAPEnvelope object or a SOAPFactory object.

For example, suppose you want to add a conformance claim header to the message to state that your message conforms to the WS-I Basic Profile. The following code fragment retrieves the SOAPHeader object from message and adds a new SOAPHeaderElement object to it. This SOAPHeaderElement object contains the correct qualified name and attribute for a WS-I conformance claim header.

```
SOAPHeader header = message.getSOAPHeader();
Name headerName = soapFactory.createName("Claim",
    "wsi", "http://ws-i.org/schemas/conformanceClaim/");
SOAPHeaderElement headerElement =
    header.addHeaderElement(headerName);
headerElement.addAttribute(soapFactory.createName(
    "conformsTo"), "http://ws-i.org/profiles/basic1.0/");
```

At this point, header contains the SOAPHeaderElement object headerElement identified by the Name object headerName. Note that the addHeaderElement method both creates headerElement and adds it to header.

A conformance claim header has no content. This code produces the following XML header:

```
<SOAP-ENV:Header>
  <wsi:Claim conformsTo="http://ws-i.org/profiles/basic1.0/"
   xmlns:wsi="http://ws-i.org/schemas/conformanceClaim/"/>
</SOAP-ENV:Header>
```

For more information about creating SOAP messages that conform to WS-I, see the Messaging section of the WS-I Basic Profile.

For a different kind of header, you might want to add content to headerElement. The following line of code uses the method addTextNode to do this.

```
headerElement.addTextNode("order");
```

Now you have the SOAPHeader object header that contains a SOAPHeaderElement object whose content is "order".

Adding Content to the SOAPPart Object

If the content you want to send is in a file, SAAJ provides an easy way to add it directly to the SOAPPart object. This means that you do not access the SOAPBody object and build the XML content yourself, as you did in the preceding section.

To add a file directly to the SOAPPart object, you use a javax.xml.transform.Source object from JAXP (the Java API for XML Processing). There are three types of Source objects: SAXSource, DOMSource, and StreamSource. A StreamSource object holds an XML document in text form. SAXSource and DOMSource objects hold content along with the instructions for transforming the content into an XML document.

The following code fragment uses the JAXP API to build a DOMSource object that is passed to the SOAPPart.setContent method. The first three lines of code get a DocumentBuilderFactory object and use it to create the DocumentBuilder object builder. Because SOAP messages use namespaces, you should set the NamespaceAware property for the factory to true. Then builder parses the content file to produce a Document object.

```
DocumentBuilderFactory dbFactory =
    DocumentBuilderFactory.newInstance();
dbFactory.setNamespaceAware(true);
DocumentBuilder builder = dbFactory.newDocumentBuilder();
Document document =
    builder.parse("file:///music/order/soap.xml");
DOMSource domSource = new DOMSource(document);
```

The following two lines of code access the SOAPPart object (using the SOAPMessage object message) and set the new Document object as its content. The SOAPPart.setContent method not only sets content for the SOAPBody object but also sets the appropriate header for the SOAPHeader object.

```
SOAPPart soapPart = message.getSOAPPart();
soapPart.setContent(domSource);
```

The XML file you use to set the content of the SOAPPart object must include Envelope and Body elements:

```
<SOAP-ENV:Envelope
xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
  ...
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

You will see other ways to add content to a message in the sections Adding a Document to the SOAP Body (page 71) and Adding Attachments (page 72).

Adding a Document to the SOAP Body

In addition to setting the content of the entire SOAP message to that of a DOMSource object, you can add a DOM document directly to the body of the message. This capability means that you do not have to create a javax.xml.transform.Source object. After you parse the document, you can add it directly to the message body:

```
SOAPBody body = message.getSOAPBody();
SOAPBodyElement docElement = body.addDocument(document);
```

Manipulating Message Content Using SAAJ or DOM APIs

Because SAAJ nodes and elements implement the DOM Node and Element interfaces, you have many options for adding or changing message content:

- Use only DOM APIs.
- Use only SAAJ APIs.
- Use SAAJ APIs and then switch to using DOM APIs.
- Use DOM APIs and then switch to using SAAJ APIs.

The first three of these cause no problems. After you have created a message, whether or not you have imported its content from another document, you can start adding or changing nodes using either SAAJ or DOM APIs.

But if you use DOM APIs and then switch to using SAAJ APIs to manipulate the document, any references to objects within the tree that were obtained using DOM APIs are no longer valid. If you must use SAAJ APIs after using DOM APIs, you should set all your DOM typed references to null, because they can become invalid. For more information about the exact cases in which references become invalid, see the SAAJ API documentation.

The basic rule is that you can continue manipulating the message content using SAAJ APIs as long as you want to, but after you start manipulating it using DOM, you should no longer use SAAJ APIs.

Adding Attachments

An AttachmentPart object can contain any type of content, including XML. And because the SOAP part can contain only XML content, you must use an AttachmentPart object for any content that is not in XML format.

Creating an AttachmentPart Object and Adding Content

The SOAPMessage object creates an AttachmentPart object, and the message also must add the attachment to itself after content has been added. The SOAPMessage class has three methods for creating an AttachmentPart object.

The first method creates an attachment with no content. In this case, an AttachmentPart method is used later to add content to the attachment.

```
AttachmentPart attachment = message.createAttachmentPart();
```

You add content to attachment by using the AttachmentPart method setContent. This method takes two parameters: a Java Object for the content, and a String object for the MIME content type that is used to encode the object. Content in the SOAPBody part of a message automatically has a Content-Type header with the value "text/xml" because the content must be in XML. In contrast, the type of content in an AttachmentPart object must be specified because it can be any type.

Each AttachmentPart object has one or more MIME headers associated with it. When you specify a type to the setContent method, that type is used for the header Content-Type. Note that Content-Type is the only header that is required. You may set other optional headers, such as Content-Id and Content-Location. For convenience, SAAJ provides get and set methods for the headers Content-Type, Content-Id, and Content-Location. These headers can be helpful in accessing a particular attachment when a message has multiple attachments. For example, to access the attachments that have particular headers, you can call the SOAPMessage method getAttachments and pass it a MIMEHeaders object containing the MIME headers you are interested in.

The following code fragment shows one of the ways to use the method setContent. The Java Object in the first parameter can be a String, a stream, a javax.xml.transform.Source object, or a javax.activation.DataHandler object. The Java Object being added in the following code fragment is a String, which is plain text, so the second argument must be "text/plain". The code also sets a content identifier, which can be used to identify this AttachmentPart object. After you have added content to attachment, you must add it to the SOAPMessage object, something that is done in the last line.

```
String stringContent = "Update address for Sunny Skies " +
   "Inc., to 10 Upbeat Street, Pleasant Grove, CA 95439";

attachment.setContent(stringContent, "text/plain");
attachment.setContentId("update_address");

message.addAttachmentPart(attachment);
```

The attachment variable now represents an AttachmentPart object that contains the string stringContent and has a header that contains the string "text/plain". It also has a

Content-Id header with "update_address" as its value. And attachment is now part of message.

The other two SOAPMessage.createAttachment methods create an AttachmentPart object complete with content. One is very similar to the AttachmentPart.setContent method in that it takes the same parameters and does essentially the same thing. It takes a Java Object containing the content and a String giving the content type. As with AttachmentPart.setContent, the Object can be a String, a stream, a javax.xml.transform.Source object, or a javax.activation.DataHandler object.

The other method for creating an AttachmentPart object with content takes a DataHandler object, which is part of the JavaBeans Activation Framework (JAF). Using a DataHandler object is fairly straightforward. First, you create a java.net.URL object for the file you want to add as content. Then you create a DataHandler object initialized with the URL object:

```
URL url = new URL("http://greatproducts.com/gizmos/img.jpg");
DataHandler dataHandler = new DataHandler(url);
AttachmentPart attachment =
    message.createAttachmentPart(dataHandler);
attachment.setContentId("attached_image");

message.addAttachmentPart(attachment);
```

You might note two things about this code fragment. First, it sets a header for Content-ID using the method setContentId. This method takes a String that can be whatever you like to identify the attachment. Second, unlike the other methods for setting content, this one does not take a String for Content-Type. This method takes care of setting the Content-Type header for you, something that is possible because one of the things a DataHandler object does is to determine the data type of the file it contains.

Accessing an AttachmentPart Object

If you receive a message with attachments or want to change an attachment to a message you are building, you need to access the attachment. The SOAPMessage class provides two versions of the getAttachments method for retrieving its AttachmentPart objects. When it is given no argument, the method SOAPMessage.getAttachments returns a java.util.Iterator object over all the AttachmentPart objects in a message. When getAttachments is given a MimeHeaders object, which is a list of MIME headers, getAttachments returns an iterator over the AttachmentPart objects that have a header that matches one of the headers in the list. The following code uses the getAttachments method that takes no arguments and thus retrieves all the

AttachmentPart objects in the SOAPMessage object message. Then it prints the content ID, the content type, and the content of each AttachmentPart object.

```
java.util.Iterator iterator = message.getAttachments();
while (iterator.hasNext()) {
    AttachmentPart attachment =
        (AttachmentPart)iterator.next();
    String id = attachment.getContentId();
    String type = attachment.getContentType();
    System.out.print("Attachment " + id +
        " has content type " + type);
    if (type == "text/plain") {
        Object content = attachment.getContent();
        System.out.println("Attachment " +
            "contains:\n" + content);
    }
}
```

Adding Attributes

An XML element can have one or more attributes that give information about that element. An attribute consists of a name for the attribute followed immediately by an equal sign (=) and its value.

The SOAPElement interface provides methods for adding an attribute, for getting the value of an attribute, and for removing an attribute. For example, in the following code fragment, the attribute named id is added to the SOAPElement object person. Because person is a SOAPElement object rather than a SOAPBodyElement object or SOAPHeaderElement object, it is legal for its Name object to contain only a local name.

```
Name attributeName = envelope.createName("id");
person.addAttribute(attributeName, "Person7");
```

These lines of code will generate the first line in the following XML fragment.

```
<person id="Person7">
  ...
</person>
```

The following line of code retrieves the value of the attribute whose name is id.

```
String attributeValue =
    person.getAttributeValue(attributeName);
```

If you had added two or more attributes to person, the preceding line of code would have returned only the value for the attribute named id. If you wanted to retrieve the values for all the attributes for person, you would use the method getAllAttributes, which returns an iterator over all the values. The following lines of code retrieve and print each value on a separate line until there are no more attribute values. Note that the Iterator.next method returns a Java Object, which is cast to a Name object so that it can be assigned to the Name object attributeName. (The examples in DOMExample.java and DOMSrcExample.java (page 95) use code similar to this.)

```
Iterator iterator = person.getAllAttributes();
while (iterator.hasNext()){
    Name attributeName = (Name) iterator.next();
    System.out.println("Attribute name is " +
        attributeName.getQualifiedName());
    System.out.println("Attribute value is " +
        element.getAttributeValue(attributeName));
}
```

The following line of code removes the attribute named id from person. The variable successful will be true if the attribute was removed successfully.

```
boolean successful = person.removeAttribute(attributeName);
```

In this section you have seen how to add, retrieve, and remove attributes. This information is general in that it applies to any element. The next section discusses attributes that can be added only to header elements.

Header Attributes

Attributes that appear in a SOAPHeaderElement object determine how a recipient processes a message. You can think of header attributes as offering a way to extend a message, giving information about such things as authentication, transaction management, payment, and so on. A header attribute refines the meaning of the header, whereas the header refines the meaning of the message contained in the SOAP body.

The SOAP 1.1 specification defines two attributes that can appear only in SOAPHeaderElement objects: actor and mustUnderstand. The next two sections discuss these attributes.

See HeaderExample.java (page 93) for an example that uses the code shown in this section.

The Actor Attribute

The actor attribute is optional, but if it is used, it must appear in a SOAPHeaderElement object. Its purpose is to indicate the recipient of a header element. The default actor is the message's ultimate recipient; that is, if no actor attribute is supplied, the message goes directly to the ultimate recipient.

An *actor* is an application that can both receive SOAP messages and forward them to the next actor. The ability to specify one or more actors as intermediate recipients makes it possible to route a message to multiple recipients and to supply header information that applies specifically to each of the recipients.

For example, suppose that a message is an incoming purchase order. Its SOAPHeader object might have SOAPHeaderElement objects with actor attributes that route the message to applications that function as the order desk, the shipping desk, the confirmation desk, and the billing department. Each of these applications will take the appropriate action, remove the SOAPHeaderElement objects relevant to it, and send the message on to the next actor.

Note: Although the SAAJ API provides the API for adding these attributes, it does not supply the API for processing them. For example, the actor attribute requires that there be an implementation such as a messaging provider service to route the message from one actor to the next.

An actor is identified by its URI. For example, the following line of code, in which orderHeader is a SOAPHeaderElement object, sets the actor to the given URI.

```
orderHeader.setActor("http://gizmos.com/orders");
```

Additional actors can be set in their own SOAPHeaderElement objects. The following code fragment first uses the SOAPMessage object message to get its SOAPHeader object header. Then header creates four SOAPHeaderElement objects, each of which sets its actor attribute.

```
SOAPHeader header = message.getSOAPHeader();
SOAPFactory soapFactory = SOAPFactory.newInstance();

String nameSpace = "ns";
String nameSpaceURI = "http://gizmos.com/NSURI";

Name order = soapFactory.createName("orderDesk",
    nameSpace, nameSpaceURI);
SOAPHeaderElement orderHeader =
    header.addHeaderElement(order);
```

```
orderHeader.setActor("http://gizmos.com/orders");

Name shipping =
    soapFactory.createName("shippingDesk",
        nameSpace, nameSpaceURI);
SOAPHeaderElement shippingHeader =
    header.addHeaderElement(shipping);
shippingHeader.setActor("http://gizmos.com/shipping");

Name confirmation =
    soapFactory.createName("confirmationDesk",
        nameSpace, nameSpaceURI);
SOAPHeaderElement confirmationHeader =
    header.addHeaderElement(confirmation);
confirmationHeader.setActor(
    "http://gizmos.com/confirmations");

Name billing = soapFactory.createName("billingDesk",
    nameSpace, nameSpaceURI);
SOAPHeaderElement billingHeader =
    header.addHeaderElement(billing);
billingHeader.setActor("http://gizmos.com/billing");
```

The SOAPHeader interface provides two methods that return a java.util.Iterator object over all the SOAPHeaderElement objects that have an actor that matches the specified actor. The first method, examineHeaderElements, returns an iterator over all the elements that have the specified actor.

```
java.util.Iterator headerElements =
    header.examineHeaderElements("http://gizmos.com/orders");
```

The second method, extractHeaderElements, not only returns an iterator over all the SOAPHeaderElement objects that have the specified actor attribute but also detaches them from the SOAPHeader object. So, for example, after the order desk application did its work, it would call extractHeaderElements to remove all the SOAPHeaderElement objects that applied to it.

```
java.util.Iterator headerElements =
    header.extractHeaderElements("http://gizmos.com/orders");
```

Each SOAPHeaderElement object can have only one actor attribute, but the same actor can be an attribute for multiple SOAPHeaderElement objects.

Two additional SOAPHeader methods—examineAllHeaderElements and extractAllHeaderElements—allow you to examine or extract all the header elements, whether or not they have an actor attribute. For example, you could use the following code to display the values of all the header elements:

```
Iterator allHeaders =
    header.examineAllHeaderElements();
while (allHeaders.hasNext()) {
    SOAPHeaderElement headerElement =
        (SOAPHeaderElement)allHeaders.next();
    Name headerName =
        headerElement.getElementName();
    System.out.println("\nHeader name is " +
        headerName.getQualifiedName());
    System.out.println("Actor is " +
        headerElement.getActor());
}
```

The mustUnderstand Attribute

The other attribute that must be added only to a SOAPHeaderElement object is mustUnderstand. This attribute says whether or not the recipient (indicated by the actor attribute) is required to process a header entry. When the value of the mustUnderstand attribute is true, the actor must understand the semantics of the header entry and must process it correctly to those semantics. If the value is false, processing the header entry is optional. A SOAPHeaderElement object with no mustUnderstand attribute is equivalent to one with a mustUnderstand attribute whose value is false.

The mustUnderstand attribute is used to call attention to the fact that the semantics in an element are different from the semantics in its parent or peer elements. This allows for robust evolution, ensuring that a change in semantics will not be silently ignored by those who may not fully understand it.

If the actor for a header that has a mustUnderstand attribute set to true cannot process the header, it must send a SOAP fault back to the sender. (See Using SOAP Faults, page 80.) The actor must not change state or cause any side effects, so that, to an outside observer, it appears that the fault was sent before any header processing was done.

The following code fragment creates a SOAPHeader object with a SOAPHeaderElement object that has a mustUnderstand attribute.

```
SOAPHeader header = message.getSOAPHeader();

Name name = soapFactory.createName("Transaction", "t",
    "http://gizmos.com/orders");
```

```
SOAPHeaderElement transaction = header.addHeaderElement(name);
transaction.setMustUnderstand(true);
transaction.addTextNode("5");
```

This code produces the following XML:

```
<SOAP-ENV:Header>
  <t:Transaction
    xmlns:t="http://gizmos.com/orders"
    SOAP-ENV:mustUnderstand="1">
   5
  </t:Transaction>
</SOAP-ENV:Header>
```

You can use the getMustUnderstand method to retrieve the value of the mustUnderstand attribute. For example, you could add the following to the code fragment at the end of the preceding section:

```
System.out.println("mustUnderstand is " +
    headerElement.getMustUnderstand());
```

Using SOAP Faults

In this section, you will see how to use the API for creating and accessing a SOAP fault element in an XML message.

Overview of SOAP Faults

If you send a message that was not successful for some reason, you may get back a response containing a SOAP fault element, which gives you status information, error information, or both. There can be only one SOAP fault element in a message, and it must be an entry in the SOAP body. Furthermore, if there is a SOAP fault element in the SOAP body, there can be no other elements in the SOAP body. This means that when you add a SOAP fault element, you have effectively completed the construction of the SOAP body.

A SOAPFault object, the representation of a SOAP fault element in the SAAJ API, is similar to an Exception object in that it conveys information about a problem. However, a SOAPFault object is quite different in that it is an element in a message's SOAPBody object rather than part of the try/catch mechanism used for Exception objects. Also, as part of the SOAPBody object, which provides a simple means for sending mandatory information intended for the ultimate recipient, a SOAP-Fault object only reports status or error information. It does not halt the execution of an application, as an Exception object can.

If you are a client using the SAAJ API and are sending point-to-point messages, the recipient of your message may add a SOAPFault object to the response to alert you to a problem. For example, if you sent an order with an incomplete address for where to send the order, the service receiving the order might put a SOAPFault object in the return message telling you that part of the address was missing.

Another example of who might send a SOAP fault is an intermediate recipient, or actor. As stated in the section Adding Attributes (page 75), an actor that cannot process a header that has a mustUnderstand attribute with a value of true must return a SOAP fault to the sender.

A SOAPFault object contains the following elements:

- A *fault code*: Always required. The fault code must be a fully qualified name: it must contain a prefix followed by a local name. The SOAP 1.1 specification defines a set of fault code local name values in section 4.4.1, which a developer can extend to cover other problems. The default fault code local names defined in the specification relate to the SAAJ API as follows:
  - VersionMismatch: The namespace for a SOAPEnvelope object was invalid.
  - MustUnderstand: An immediate child element of a SOAPHeader object had its mustUnderstand attribute set to true, and the processing party did not understand the element or did not obey it.
  - Client: The SOAPMessage object was not formed correctly or did not contain the information needed to succeed.
  - Server: The SOAPMessage object could not be processed because of a processing error, not because of a problem with the message itself.
- A *fault string*: Always required. A human-readable explanation of the fault.
- A *fault actor*: Required if the SOAPHeader object contains one or more actor attributes; optional if no actors are specified, meaning that the only actor is the ultimate destination. The fault actor, which is specified as a URI, identifies who caused the fault. For an explanation of what an actor is, see The Actor Attribute, page 77.
- A *Detail object*: Required if the fault is an error related to the SOAPBody object. If, for example, the fault code is Client, indicating that the message could not be processed because of a problem in the SOAPBody object, the

SOAPFault object must contain a Detail object that gives details about the problem. If a SOAPFault object does not contain a Detail object, it can be assumed that the SOAPBody object was processed successfully.

Creating and Populating a SOAPFault Object

You have seen how to add content to a SOAPBody object; this section walks you through adding a SOAPFault object to a SOAPBody object and then adding its constituent parts.

As with adding content, the first step is to access the SOAPBody object.

```
SOAPBody body = message.getSOAPBody();
```

With the SOAPBody object body in hand, you can use it to create a SOAPFault object. The following line of code creates a SOAPFault object and adds it to body.

```
SOAPFault fault = body.addFault();
```

The SOAPFault interface provides convenience methods that create an element, add the new element to the SOAPFault object, and add a text node, all in one operation. For example, in the following lines of code, the method setFaultCode creates a faultcode element, adds it to fault, and adds a Text node with the value "SOAP-ENV:Server" by specifying a default prefix and the namespace URI for a SOAP envelope.

```
Name faultName =
    soapFactory.createName("Server",
        "", SOAPConstants.URI_NS_SOAP_ENVELOPE);
fault.setFaultCode(faultName);
fault.setFaultActor("http://gizmos.com/orders");
fault.setFaultString("Server not responding");
```

The SOAPFault object fault, created in the preceding lines of code, indicates that the cause of the problem is an unavailable server and that the actor at http://gizmos.com/orders is having the problem. If the message were being routed only to its ultimate destination, there would have been no need to set a fault actor. Also note that fault does not have a Detail object because it does not relate to the SOAPBody object.

The following code fragment creates a SOAPFault object that includes a Detail object. Note that a SOAPFault object can have only one Detail object, which is simply a container for DetailEntry objects, but the Detail object can have multiple

DetailEntry objects. The Detail object in the following lines of code has two DetailEntry objects added to it.

```
SOAPFault fault = body.addFault();

Name faultName = soapFactory.createName("Client",
    "", SOAPConstants.URI_NS_SOAP_ENVELOPE);
fault.setFaultCode(faultName);
fault.setFaultString("Message does not have necessary info");

Detail detail = fault.addDetail();

Name entryName = soapFactory.createName("order",
    "PO", "http://gizmos.com/orders/");
DetailEntry entry = detail.addDetailEntry(entryName);
entry.addTextNode("Quantity element does not have a value");

Name entryName2 = soapFactory.createName("confirmation",
    "PO", "http://gizmos.com/confirm");
DetailEntry entry2 = detail.addDetailEntry(entryName2);
entry2.addTextNode("Incomplete address: no zip code");
```

See SOAPFaultTest.java (page 101) for an example that uses code like that shown in this section.

Retrieving Fault Information

Just as the SOAPFault interface provides convenience methods for adding information, it also provides convenience methods for retrieving that information. The following code fragment shows what you might write to retrieve fault information from a message you received. In the code fragment, newMessage is the SOAPMessage object that has been sent to you. Because a SOAPFault object must be part of the SOAPBody object, the first step is to access the SOAPBody object. Then the code tests to see whether the SOAPBody object contains a SOAPFault object. If it does, the code retrieves the SOAPFault object and uses it to retrieve its contents. The convenience methods getFaultCode, getFaultString, and getFaultActor make retrieving the values very easy.

```
SOAPBody body = newMessage.getSOAPBody();
if ( body.hasFault() ) {
    SOAPFault newFault = body.getFault();
    Name code = newFault.getFaultCodeAsName();
    String string = newFault.getFaultString();
    String actor = newFault.getFaultActor();
```

Next the code prints the values it has just retrieved. Not all messages are required to have a fault actor, so the code tests to see whether there is one. Testing whether the variable actor is null works because the method getFaultActor returns null if a fault actor has not been set.

```
System.out.println("SOAP fault contains: ");
System.out.println("  Fault code = " +
    code.getQualifiedName());
System.out.println("  Fault string = " + string);

if ( actor != null ) {
    System.out.println("  Fault actor = " + actor);
}
```

The final task is to retrieve the Detail object and get its DetailEntry objects. The code uses the SOAPFault object newFault to retrieve the Detail object newDetail, and then it uses newDetail to call the method getDetailEntries. This method returns the java.util.Iterator object entries, which contains all the DetailEntry objects in newDetail. Not all SOAPFault objects are required to have a Detail object, so the code tests to see whether newDetail is null. If it is not, the code prints the values of the DetailEntry objects as long as there are any.

```
Detail newDetail = newFault.getDetail();
if (newDetail != null) {
    Iterator entries = newDetail.getDetailEntries();
    while ( entries.hasNext() ) {
        DetailEntry newEntry =
            (DetailEntry)entries.next();
        String value = newEntry.getValue();
        System.out.println("  Detail entry = " + value);
    }
}
```

In summary, you have seen how to add a SOAPFault object and its contents to a message as well as how to retrieve the contents. A SOAPFault object, which is optional, is added to the SOAPBody object to convey status or error information. It must always have a fault code and a String explanation of the fault. A SOAPFault object must indicate the actor that is the source of the fault only when there are multiple actors; otherwise, it is optional. Similarly, the SOAPFault object must contain a Detail object with one or more DetailEntry objects only when the contents of the SOAPBody object could not be processed successfully.

See SOAPFaultTest.java (page 101) for an example that uses code like that shown in this section.

Code Examples

The first part of this tutorial uses code fragments to walk you through the fundamentals of using the SAAJ API. In this section, you will use some of those code fragments to create applications. First, you will see the program Request.java. Then you will see how to run the programs MyUddiPing.java, HeaderExample.java, DOMExample.java, DOMSrcExample.java, Attachments.java, and SOAPFaultTest.java.

To run these examples, you will deploy them to the Sun Java System Application Server Platform 8.1 from the IDE.

Request.java

The class Request.java puts together the code fragments used in the section Tutorial (page 59) and adds what is needed to make it a complete example of a client sending a request-response message. In addition to putting all the code together, it adds import statements, a main method, and a try/catch block with exception handling.

```
import javax.xml.soap.*;
import java.util.*;
import java.net.URL;

public class Request {
    public static void main(String[] args){
        try {
            SOAPConnectionFactory soapConnectionFactory =
                SOAPConnectionFactory.newInstance();
            SOAPConnection connection =
                soapConnectionFactory.createConnection();
            SOAPFactory soapFactory =
                SOAPFactory.newInstance();

            MessageFactory factory =
                MessageFactory.newInstance();
            SOAPMessage message = factory.createMessage();

            SOAPHeader header = message.getSOAPHeader();
            SOAPBody body = message.getSOAPBody();
            header.detachNode();

            Name bodyName = soapFactory.createName(
                "GetLastTradePrice", "m",
                "http://wombats.ztrade.com");
            SOAPBodyElement bodyElement =
                body.addBodyElement(bodyName);

            Name name = soapFactory.createName("symbol");
            SOAPElement symbol =
                bodyElement.addChildElement(name);
            symbol.addTextNode("SUNW");

            URL endpoint = new URL
                ("http://wombat.ztrade.com/quotes");
            SOAPMessage response =
                connection.call(message, endpoint);

            connection.close();

            SOAPBody soapBody = response.getSOAPBody();

            Iterator iterator =
                soapBody.getChildElements(bodyName);
            bodyElement = (SOAPBodyElement)iterator.next();
            String lastPrice = bodyElement.getValue();

            System.out.print("The last price for SUNW is ");
            System.out.println(lastPrice);

        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

For Request.java to be runnable, the second argument supplied to the call method would have to be a valid existing URI, and this is not true in this case. However, the application in the next section is one that you can run.

MyUddiPing.java

The program MyUddiPing.java is another example of a SAAJ client application. It sends a request to a Universal Description, Discovery and Integration (UDDI) service and gets back the response. A UDDI service is a business registry and repository from which you can get information about businesses that have registered themselves with the registry service. For this example, the MyUddiPing application is not actually accessing a UDDI service registry but rather a test (demo) version. Because of this, the number of businesses you can get information about is limited. Nevertheless, MyUddiPing demonstrates a request being sent and a response being received.

Setting Up

The MyUddiPing example is in the following directory:

```
<INSTALL>/j2eetutorial14/examples/saaj/myuddiping/
```

Note: *<INSTALL>* is the directory where you installed the tutorial bundle.

In the myuddiping directory, you will find an IDE project called MyUddiPing. Its src directory contains one source file, MyUddiPing.java.

The file uddi.properties contains the URL of the destination (a UDDI test registry) and the proxy host and proxy port of the sender. By default, the destination is the IBM test registry; the Microsoft test registry is commented out.

If you access the Internet from behind a firewall, edit the uddi.properties file to supply the correct proxy host and proxy port. If you are not sure what the values for these are, consult your system administrator or another person with that information. The typical value of the proxy port is 8080. You can also edit the file to specify another registry.

The file build.xml is the IDE's build file for this example. The Build Project command is hooked up to a target in the build.xml file that compiles the source file MyUddiPing.java and puts the resulting .class file in the build directory. So to do these tasks, you take the same steps as above and right-click the project, after which you choose Build Project.

Examining MyUddiPing

We will go through the file MyUddiPing.java a few lines at a time, concentrating on the last section. This is the part of the application that accesses only the content you want from the XML message returned by the UDDI registry.

The first few lines of code import the packages used in the application.

```
import javax.xml.soap.*;
import java.net.*;
import java.util.*;
import java.io.*;
```

The next few lines begin the definition of the class MyUddiPing, which starts with the definition of its main method. The first thing it does is to check to see whether two arguments were supplied. If they were not, it prints a usage message and exits. The usage message mentions only one argument; the other is supplied by the build.xml target.

```
public class MyUddiPing {
    public static void main(String[] args) {
        try {
            if (args.length != 2) {
                System.err.println("Usage: asant run " +
                    "-Dbusiness-name=<name>");
                System.exit(1);
            }
```

The following lines create a java.util.Properties object that contains the system properties and the properties from the file uddi.properties, which is in the myuddiping directory.

```
            Properties myprops = new Properties();
            myprops.load(new FileInputStream(args[0]));

            Properties props = System.getProperties();

            Enumeration propNames = myprops.propertyNames();
            while (propNames.hasMoreElements()) {
                String s = (String)propNames.nextElement();
                props.setProperty(s, myprops.getProperty(s));
            }
```

The next four lines create a SOAPMessage object. First, the code gets an instance of SOAPConnectionFactory and uses it to create a connection. Then it gets an instance of MessageFactory and an instance of SOAPFactory, using the MessageFactory instance to create a message.

```
            SOAPConnectionFactory soapConnectionFactory =
                SOAPConnectionFactory.newInstance();
            SOAPConnection connection =
                soapConnectionFactory.createConnection();
            MessageFactory messageFactory =
                MessageFactory.newInstance();
            SOAPFactory soapFactory = SOAPFactory.newInstance();

            SOAPMessage message =
                messageFactory.createMessage();
```

The next lines of code retrieve the SOAPHeader and SOAPBody objects from the message and remove the header.

```
SOAPHeader header = message.getSOAPHeader();
SOAPBody body = message.getSOAPBody();
header.detachNode();
```

The following lines of code create the UDDI find_business message. The first line creates a SOAPBodyElement with a fully qualified name, including the required namespace for a UDDI version 2 message. The next lines add two attributes to the new element: the required attribute generic, with the UDDI version number 2.0, and the optional attribute maxRows, with the value 100. Then the code adds a child element that has the Name object name and adds text to the element by using the method addTextNode. The added text is the business name you will supply at the command line when you run the application.

```
SOAPBodyElement findBusiness =
    body.addBodyElement(soapFactory.createName(
        "find_business", "",
        "urn:uddi-org:api_v2"));
findBusiness.addAttribute(soapFactory.createName(
    "generic"), "2.0");
findBusiness.addAttribute(soapFactory.createName(
    "maxRows"), "100");

SOAPElement businessName =
    findBusiness.addChildElement(
        soapFactory.createName("name"));
businessName.addTextNode(args[1]);
```

The next line of code saves the changes that have been made to the message. This method will be called automatically when the message is sent, but it does not hurt to call it explicitly.

```
message.saveChanges();
```

The following lines display the message that will be sent:

```
System.out.println("\n--- Request Message ---\n");
message.writeTo(System.out);
```

The next line of code creates the java.net.URL object that represents the destination for this message. It gets the value of the property named URL from the system property file.

```
URL endpoint = new URL(
    System.getProperties().getProperty("URL"));
```

Next, the message message is sent to the destination that endpoint represents, which is the UDDI test registry. The call method will block until it gets a SOAPMessage object back, at which point it returns the reply.

```
SOAPMessage reply =
    connection.call(message, endpoint);
```

In the next lines of code, the first line prints a line giving the URL of the sender (the test registry), and the others display the returned message.

```
System.out.println("\n\nReceived reply from: " +
    endpoint);
System.out.println("\n---- Reply Message ----\n");
reply.writeTo(System.out);
```

The returned message is the complete SOAP message, an XML document, as it looks when it comes over the wire. It is a businessList that follows the format specified in http://uddi.org/pubs/DataStructure-V2.03-Published-20020719.htm#_Toc25130802.

As interesting as it is to see the XML that is actually transmitted, the XML document format does not make it easy to see the text that is the message's content. To remedy this, the last part of MyUddiPing.java contains code that prints only the text content of the response, making it much easier to see the information you want.

Because the content is in the SOAPBody object, the first step is to access it, as shown in the following line of code.

```
SOAPBody replyBody = reply.getSOAPBody();
```

Next, the code displays a message describing the content:

```
System.out.println("\n\nContent extracted from " +
    "the reply message:\n");
```

To display the content of the message, the code uses the known format of the reply message. First, it gets all the reply body's child elements named businessList:

```
Iterator businessListIterator =
    replyBody.getChildElements(
        soapFactory.createName("businessList",
            "", "urn:uddi-org:api_v2"));
```

The method getChildElements returns the elements in the form of a java.util.Iterator object. You access the child elements by calling the method next on the Iterator object. An immediate child of a SOAPBody object is a SOAPBodyElement object.

We know that the reply can contain only one businessList element, so the code then retrieves this one element by calling the iterator's next method. Note that the method Iterator.next returns an Object, which must be cast to the specific kind of object you are retrieving. Thus, the result of calling businessListIterator.next is cast to a SOAPBodyElement object:

```
SOAPBodyElement businessList =
    (SOAPBodyElement)businessListIterator.next();
```

The next element in the hierarchy is a single businessInfos element, so the code retrieves this element in the same way it retrieved the businessList. Children of SOAPBodyElement objects and all child elements from this point forward are SOAPElement objects.

```
Iterator businessInfosIterator =
    businessList.getChildElements(
        soapFactory.createName("businessInfos",
            "", "urn:uddi-org:api_v2"));

SOAPElement businessInfos =
    (SOAPElement)businessInfosIterator.next();
```

The businessInfos element contains zero or more businessInfo elements. If the query returned no businesses, the code prints a message saying that none were found. If the query returned businesses, however, the code extracts the name and optional description by retrieving the child elements that have those names. The method Iterator.hasNext can be used in a while loop because it returns true as long as the next call to the method next will return a child element. Accordingly, the loop ends when there are no more child elements to retrieve.

```
Iterator businessInfoIterator =
    businessInfos.getChildElements(
        soapFactory.createName("businessInfo",
            "", "urn:uddi-org:api_v2"));

if (! businessInfoIterator.hasNext()) {
    System.out.println("No businesses found " +
        "matching the name \"" + args[1] + "\".");
} else {
    while (businessInfoIterator.hasNext()) {
        SOAPElement businessInfo = (SOAPElement)
            businessInfoIterator.next();

        Iterator nameIterator =
            businessInfo.getChildElements(
                soapFactory.createName("name",
                    "", "urn:uddi-org:api_v2"));
        while (nameIterator.hasNext()) {
            businessName =
                (SOAPElement)nameIterator.next();
            System.out.println("Company name: " +
                businessName.getValue());
        }
        Iterator descriptionIterator =
            businessInfo.getChildElements(
                soapFactory.createName(
                    "description", "",
                    "urn:uddi-org:api_v2"));
        while (descriptionIterator.hasNext()) {
            SOAPElement businessDescription =
                (SOAPElement) descriptionIterator.next();
            System.out.println("Description: " +
                businessDescription.getValue());
        }
        System.out.println("");
    }
```

Running MyUddiPing

Make sure you have edited the uddi.properties file and compiled MyUddiPing.java as described in Setting Up (page 87).

To run the application, follow these steps:

1. If you have not already opened the MyUddiPing project, choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to *<INSTALL>*/j2eetutorial14/examples/saaj/, select the myuddiping project, and choose Open Project Folder.
2. The project needs to know the location of some JAR files on its classpath. Right-click the project and choose Resolve Reference Problems. Select the "activation.jar" file/folder could not be found message and click Resolve. In the file chooser, select navigate to the lib directory in your application server installation, select activation.jar, and click OK. The IDE automatically resolves the location of the other missing JAR files. Click Close.
3. Right-click the project in the Projects window, choose Properties, click Run, and type uddi.properties food in the Arguments field. The first argument is the file uddi.properties. The other argument is the name of the business for which you want to get a description. Click OK.
4. In the Projects window, right-click the project and choose Run Project.
5. In the Output window, the application displays the following output:

```
Content extracted from the reply message:

Company name: Food
Description: Test Food

Company name: Food Manufacturing

Company name: foodCompanyA
Description: It is a food company sells biscuit
```

If you want to run MyUddiPing again, you may want to start over by deleting the build directory and the .class file it contains. You can do this by right-clicking the project node in the Projects window and choosing Clean Project.

HeaderExample.java

The example HeaderExample.java, based on the code fragments in the section Adding Attributes (page 75), creates a message that has several headers. It then retrieves the contents of the headers and prints them. You will find the code for HeaderExample in the following directory:

```
<INSTALL>/j2eetutorial14/examples/saaj/headerexample
```

Running HeaderExample

To run the application, follow these steps:

1. If you have not already opened the HeaderExample project, choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to *<INSTALL>*/j2eetutorial14/examples/saaj/, select the headerexample project, and choose Open Project Folder.
2. The project needs to know the location of some JAR files on its classpath. Right-click the project and choose Resolve Reference Problems. Select the “activation.jar” file/folder could not be found message and click Resolve. In the file chooser, select navigate to the lib directory in your application server installation, select activation.jar, and click OK. The IDE automatically resolves the location of the other missing JAR files. Click Close.
3. In the Projects window, right-click the project and choose Run Project.
4. In the Output window, the application displays the following output:

```
----- Request Message ----

<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header>
<ns:orderDesk SOAP-ENV:actor="http://gizmos.com/orders" xmlns:ns="http://gizmos.com/
NSURI"/>
<ns:shippingDesk SOAP-ENV:actor="http://gizmos.com/shipping" xmlns:ns="http://
gizmos.com/NSURI"/>
<ns:confirmationDesk
SOAP-ENV:actor="http://gizmos.com/confirmations" xmlns:ns="http://gizmos.com/
NSURI"/>
<ns:billingDesk SOAP-ENV:actor="http://gizmos.com/billing" xmlns:ns="http://
gizmos.com/NSURI"/>
<t:Transaction SOAP-ENV:mustUnderstand="1" xmlns:t="http://gizmos.com/orders">5</
t:Transaction>
</SOAP-ENV:Header><SOAP-ENV:Body/></SOAP-ENV:Envelope>
Header name is ns:orderDesk
Actor is http://gizmos.com/orders
mustUnderstand is false

Header name is ns:shippingDesk
Actor is http://gizmos.com/shipping
mustUnderstand is false

Header name is ns:confirmationDesk
Actor is http://gizmos.com/confirmations
mustUnderstand is false
```

```
Header name is ns:billingDesk
Actor is http://gizmos.com/billing
mustUnderstand is false

Header name is t:Transaction
Actor is null
mustUnderstand is true
```

DOMExample.java and DOMSrcExample.java

The examples DOMExample.java and DOMSrcExample.java show how to add a DOM document to a message and then traverse its contents. They show two ways to do this:

- DOMExample.java creates a DOM document and adds it to the body of a message.
- DOMSrcExample.java creates the document, uses it to create a DOMSource object, and then sets the DOMSource object as the content of the message's SOAP part.

You will find the code for DOMExample and DOMSrcExample in the following directory:

```
<INSTALL>/j2eetutorial14/examples/saaj/dom
```

Examining DOMExample

DOMExample first creates a DOM document by parsing an XML document. The file it parses is one that you specify on the command line.

```
static Document document;
...
    DocumentBuilderFactory factory =
        DocumentBuilderFactory.newInstance();
    factory.setNamespaceAware(true);
    try {
        DocumentBuilder builder = factory.newDocumentBuilder();
        document = builder.parse( new File(args[0]) );
        ...
```

Next, the example creates a SOAP message in the usual way. Then it adds the document to the message body:

```
SOAPBodyElement docElement = body.addDocument(document);
```

This example does not change the content of the message. Instead, it displays the message content and then uses a recursive method, getContents, to traverse the element tree using SAAJ APIs and display the message contents in a readable form.

```
public void getContents(Iterator iterator, String indent) {

  while (iterator.hasNext()) {
    Node node = (Node) iterator.next();
    SOAPElement element = null;
    Text text = null;
    if (node instanceof SOAPElement) {
      element = (SOAPElement)node;
      Name name = element.getElementName();
      System.out.println(indent + "Name is " +
        name.getQualifiedName());
      Iterator attrs = element.getAllAttributes();
      while (attrs.hasNext()){
        Name attrName = (Name)attrs.next();
        System.out.println(indent +
          " Attribute name is " +
          attrName.getQualifiedName());
        System.out.println(indent +
          " Attribute value is " +
          element.getAttributeValue(attrName));
      }
      Iterator iter2 = element.getChildElements();
      getContents(iter2, indent + " ");
    } else {
      text = (Text) node;
      String content = text.getValue();
      System.out.println(indent +
        "Content is: " + content);
      }
    }
  }
```

Examining DOMSrcExample

DOMSrcExample differs from DOMExample in only a few ways. First, after it parses the document, DOMSrcExample uses the document to create a DOMSource object. This code is the same as that of DOMExample except for the last line:

```
static DOMSource domSource;
...
try {
    DocumentBuilder builder =
        factory.newDocumentBuilder();
    document = builder.parse(new File(args[0]));
    domSource = new DOMSource(document);
    ...
```

Then, after DOMSrcExample creates the message, it does not get the header and body and add the document to the body, as DOMExample does. Instead, DOMSrcExample gets the SOAP part and sets the DOMSource object as its content:

```
// Create a message
SOAPMessage message = messageFactory.createMessage();

// Get the SOAP part and set its content to domSource
SOAPPart soapPart = message.getSOAPPart();
soapPart.setContent(domSource);
```

The example then uses the getContents method to obtain the contents of both the header (if it exists) and the body of the message.

The most important difference between these two examples is the kind of document you can use to create the message. Because DOMExample adds the document to the body of the SOAP message, you can use any valid XML file to create the document. But because DOMSrcExample makes the document the entire content of the message, the document must already be in the form of a valid SOAP message, and not just any XML document.

Running DOMExample and DOMSrcExample

To run DOMExample and DOMSrcExample, you use the IDE project that is in the directory <*INSTALL*>/j2eetutorial14/examples/saaj. This directory also contains several sample XML files you can use:

- domsrc1.xml, an example that has a SOAP header (the contents of the HeaderExample output) and the body of a UDDI query
- domsrc2.xml, an example of a reply to a UDDI query (specifically, some sample output from the MyUddiPing example), but with spaces added for readability
- uddimsg.xml, similar to domsrc2.xml except that it is only the body of the message and contains no spaces
- slide.xml

To run the application, follow these steps:

1. If you have not already opened the DomExample project, choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to <*INSTALL*>/j2eetutorial14/examples/saaj/, select the DomExample project, and choose Open Project Folder.
2. The project needs to know the location of some JAR files on its classpath. Right-click the project and choose Resolve Reference Problems. Select the “saaj-api.jar” file/folder could not be found message and click Resolve. In the file chooser, select navigate to the lib directory in your application server installation, select saaj-api.jar, and click OK. The IDE automatically resolves the location of the other missing JAR files. Click Close.
3. Right-click the project in the Projects window, choose Properties, click Run, and type domsrc1.xml (or any of the other arguments above). Click OK.
4. In the Projects window, right-click the project and choose Run Project.
5. In the Output window, the application displays the following output:

```
Running DOMExample.
Name is businessList
Attribute name is generic
Attribute value is 2.0
Attribute name is operator
Attribute value is www.ibm.com/services/uddi
Attribute name is truncated
Attribute value is false
Attribute name is xmlns
Attribute value is urn:uddi-org:api_v2
...
```

To run DOMSrcExample, first right-click the project in the Projects window, choose Properties, click Run, and type domexample.DOMSrcExample in the Main Class field and domsrc2.xml in the Arguments field. Then right-click the project and choose Run Project.

When you run DOMSrcExample, you will see output that begins like the following:

```
run-domsrc:
    Running DOMSrcExample.
    Body contents:
    Content is:

    Name is businessList
     Attribute name is generic
     Attribute value is 2.0
     Attribute name is operator
     Attribute value is www.ibm.com/services/uddi
     Attribute name is truncated
     Attribute value is false
     Attribute name is xmlns
     Attribute value is urn:uddi-org:api_v2
     ...
```

If you run DOMSrcExample with the file uddimsg.xml or slide.xml, you will see runtime errors.

Attachments.java

The example Attachments.java, based on the code fragments in the sections Creating an AttachmentPart Object and Adding Content (page 72) and Accessing an AttachmentPart Object (page 74), creates a message that has a text attachment and an image attachment. It then retrieves the contents of the attachments and prints the contents of the text attachment. You will find the code for Attachments in the following directory:

```
<INSTALL>/j2eetutorial14/examples/saaj/attachments/
```

The Attachments.java program first creates a message in the usual way. It then creates an AttachmentPart for the text attachment:

```
AttachmentPart attachment1 = message.createAttachmentPart();
```

After it reads input from a file into a string named stringContent, it sets the content of the attachment to the value of the string and the type to text/plain and also sets a content ID.

```
attachment1.setContent(stringContent, "text/plain");
attachment1.setContentId("attached_text");
```

It then adds the attachment to the message:

```
message.addAttachmentPart(attachment1);
```

The example uses a javax.activation.DataHandler object to hold a reference to the graphic that constitutes the second attachment. It creates this attachment using the form of the createAttachmentPart method that takes a DataHandler argument.

```
// Create attachment part for image
URL url = new URL("file:///../xml-pic.jpg");
DataHandler dataHandler = new DataHandler(url);
AttachmentPart attachment2 =
    message.createAttachmentPart(dataHandler);
attachment2.setContentId("attached_image");

message.addAttachmentPart(attachment2);
```

The example then retrieves the attachments from the message. It displays the contentId and contentType attributes of each attachment and the contents of the text attachment.

Running Attachments

To run the application, follow these steps:

1. If you have not already opened the Attachments project, choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to *<INSTALL>*/j2eetutorial14/examples/saaj/, select the Attachments project, and choose Open Project Folder.
2. The project needs to know the location of some JAR files on its classpath. Right-click the project and choose Resolve Reference Problems. Select the

“activation.jar” file/folder could not be found message and click Resolve. In the file chooser, select navigate to the lib directory in your application server installation, select activation.jar, and click OK. The IDE automatically resolves the location of the other missing JAR files. Click Close.

3. Right-click the project in the Projects window, choose Properties, click Run, and type addr.txt in the Arguments field. This file is included with the application. Click OK.
4. In the Projects window, right-click the project and choose Run Project.
5. In the Output window, the application displays the following output:

```
run:
Attachment attached_text has content type text/plain
Attachment contains:
Update address for Sunny Skies, Inc., to
10 Upbeat Street
Pleasant Grove, CA 95439

Attachment attached_image has content type image/jpeg
```

SOAPFaultTest.java

The example SOAPFaultTest.java, based on the code fragments in the sections Creating and Populating a SOAPFault Object (page 82) and Retrieving Fault Information (page 83), creates a message that has a SOAPFault object. It then retrieves the contents of the SOAPFault object and prints them. You will find the code for SOAPFaultTest in the following directory:

```
<INSTALL>/j2eetutorial14/examples/saaj/soapfaulttest/
```

Running SOAPFaultTest

To run the application, follow these steps:

1. If you have not already opened the Attachments project, choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to *<INSTALL>*/j2eetutorial14/examples/saaj/, select the soapfaulttest project, and choose Open Project Folder.
2. The project needs to know the location of some JAR files on its classpath. Right-click the project and choose Resolve Reference Problems. Select the “activation.jar” file/folder could not be found message and click Resolve. In the file chooser, select navigate to the lib directory in your application server

installation, select activation.jar, and click OK. The IDE automatically resolves the location of the other missing JAR files. Click Close.

3. In the Projects window, right-click the project and choose Run Project.
4. In the Output window, the application displays the following output (line breaks have been inserted in the message for readability):

```
Here is what the XML message looks like:

<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header/><SOAP-ENV:Body>
<SOAP-ENV:Fault><faultcode>SOAP-ENV:Client</faultcode>
<faultstring>Message does not have necessary info</faultstring>
<faultactor>http://gizmos.com/order</faultactor>
<detail>
<PO:order xmlns:PO="http://gizmos.com/orders/">
Quantity element does not have a value</PO:order>
<PO:confirmation xmlns:PO="http://gizmos.com/confirm">
Incomplete address: no zip code</PO:confirmation>
</detail></SOAP-ENV:Fault>
</SOAP-ENV:Body></SOAP-ENV:Envelope>

SOAP fault contains:
  Fault code = SOAP-ENV:Client
  Local name = Client
  Namespace prefix = SOAP-ENV, bound to
http://schemas.xmlsoap.org/soap/envelope/
  Fault string = Message does not have necessary info
  Fault actor = http://gizmos.com/order
  Detail entry = Quantity element does not have a value
  Detail entry = Incomplete address: no zip code
```

Further Information

For more information about SAAJ, SOAP, and WS-I, see the following:

- SAAJ 1.2 specification, available from
  http://java.sun.com/xml/downloads/saaj.html
- SAAJ web site:
  http://java.sun.com/xml/saaj/
- WS-I Basic Profile:

http://www.ws-i.org/Profiles/Basic/2003-08/
BasicProfile-1.0a.html

- JAXM web site:

  http://java.sun.com/xml/jaxm/

4 Enterprise Beans

ENTERPRISE beans are the J2EE components that implement Enterprise JavaBeans (EJB) technology. Enterprise beans run in the EJB container, a runtime environment within the Sun Java System Application Server Platform Edition 8 (see Figure 1–5, page 10). Although transparent to the application developer, the EJB container provides system-level services such as transactions and security to its enterprise beans. These services enable you to quickly build and deploy enterprise beans, which form the core of transactional J2EE applications.

What Is an Enterprise Bean?

Written in the Java programming language, an *enterprise bean* is a server-side component that encapsulates the business logic of an application. The business logic is the code that fulfills the purpose of the application. In an inventory control application, for example, the enterprise beans might implement the business logic in methods called checkInventoryLevel and orderProduct. By invoking these methods, remote clients can access the inventory services provided by the application.

Benefits of Enterprise Beans

For several reasons, enterprise beans simplify the development of large, distributed applications. First, because the EJB container provides system-level services to enterprise beans, the bean developer can concentrate on solving business problems. The EJB container—and not the bean developer—is responsible for system-level services such as transaction management and security authorization.

Second, because the beans—and not the clients—contain the application's business logic, the client developer can focus on the presentation of the client. The client developer does not have to code the routines that implement business rules or access databases. As a result, the clients are thinner, a benefit that is particularly important for clients that run on small devices.

Third, because enterprise beans are portable components, the application assembler can build new applications from existing beans. These applications can run on any compliant J2EE server provided that they use the standard APIs.

When to Use Enterprise Beans

You should consider using enterprise beans if your application has any of the following requirements:

- The application must be scalable. To accommodate a growing number of users, you may need to distribute an application's components across multiple machines. Not only can the enterprise beans of an application run on different machines, but also their location will remain transparent to the clients.
- Transactions must ensure data integrity. Enterprise beans support transactions, the mechanisms that manage the concurrent access of shared objects.
- The application will have a variety of clients. With only a few lines of code, remote clients can easily locate enterprise beans. These clients can be thin, various, and numerous.

Types of Enterprise Beans

Table 4–1 summarizes the three types of enterprise beans. The following sections discuss each type in more detail.

Table 4–1 Enterprise Bean Types

Enterprise Bean Type	Purpose
Session	Performs a task for a client; implements a web service
Entity	Represents a business entity object that exists in persistent storage
Message-Driven	Acts as a listener for the Java Message Service API, processing messages asynchronously

What Is a Session Bean?

A *session bean* represents a single client inside the Application Server. To access an application that is deployed on the server, the client invokes the session bean’s methods. The session bean performs work for its client, shielding the client from complexity by executing business tasks inside the server.

As its name suggests, a session bean is similar to an interactive session. A session bean is not shared; it can have only one client, in the same way that an interactive session can have only one user. Like an interactive session, a session bean is not persistent. (That is, its data is not saved to a database.) When the client terminates, its session bean appears to terminate and is no longer associated with the client.

For code samples, see Chapter 6.

State Management Modes

There are two types of session beans: stateless and stateful.

Stateless Session Beans

A *stateless* session bean does not maintain a conversational state for the client. When a client invokes the method of a stateless bean, the bean's instance variables may contain a state, but only for the duration of the invocation. When the method is finished, the state is no longer retained. Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client.

Because stateless session beans can support multiple clients, they can offer better scalability for applications that require large numbers of clients. Typically, an application requires fewer stateless session beans than stateful session beans to support the same number of clients.

At times, the EJB container may write a stateful session bean to secondary storage. However, stateless session beans are never written to secondary storage. Therefore, stateless beans may offer better performance than stateful beans.

A stateless session bean can implement a web service, but other types of enterprise beans cannot.

Stateful Session Beans

The state of an object consists of the values of its instance variables. In a *stateful* session bean, the instance variables represent the state of a unique client-bean session. Because the client interacts ("talks") with its bean, this state is often called the *conversational state*.

The state is retained for the duration of the client-bean session. If the client removes the bean or terminates, the session ends and the state disappears. This transient nature of the state is not a problem, however, because when the conversation between the client and the bean ends there is no need to retain the state.

When to Use Session Beans

In general, you should use a session bean if the following circumstances hold:

- At any given time, only one client has access to the bean instance.
- The state of the bean is not persistent, existing only for a short period (perhaps a few hours).
- The bean implements a web service.

Stateful session beans are appropriate if any of the following conditions are true:

- The bean's state represents the interaction between the bean and a specific client.
- The bean needs to hold information about the client across method invocations.
- The bean mediates between the client and the other components of the application, presenting a simplified view to the client.
- Behind the scenes, the bean manages the work flow of several enterprise beans. For an example, see the AccountControllerBean session bean in Chapter 36.

To improve performance, you might choose a stateless session bean if it has any of these traits:

- The bean's state has no data for a specific client.
- In a single method invocation, the bean performs a generic task for all clients. For example, you might use a stateless session bean to send an email that confirms an online order.
- The bean fetches from a database a set of read-only data that is often used by clients. Such a bean, for example, could retrieve the table rows that represent the products that are on sale this month.

What Is an Entity Bean?

An *entity bean* represents a business object in a persistent storage mechanism. Some examples of business objects are customers, orders, and products. In the Application Server, the persistent storage mechanism is a relational database. Typically, each entity bean has an underlying table in a relational database, and each instance of the bean corresponds to a row in that table. For code examples of entity beans, please refer to Chapters 7 and 8.

What Makes Entity Beans Different from Session Beans?

Entity beans differ from session beans in several ways. Entity beans are persistent, allow shared access, have primary keys, and can participate in relationships with other entity beans.

Persistence

Because the state of an entity bean is saved in a storage mechanism, it is persistent. *Persistence* means that the entity bean's state exists beyond the lifetime of the application or the Application Server process. If you've worked with databases, you're familiar with persistent data. The data in a database is persistent because it still exists even after you shut down the database server or the applications it services.

There are two types of persistence for entity beans: bean-managed and container-managed. With *bean-managed* persistence, the entity bean code that you write contains the calls that access the database. If your bean has *container-managed* persistence, the EJB container automatically generates the necessary database access calls. The code that you write for the entity bean does not include these calls. For additional information, see the section Container-Managed Persistence (page 111).

Shared Access

Entity beans can be shared by multiple clients. Because the clients might want to change the same data, it's important that entity beans work within transactions. Typically, the EJB container provides transaction management. In this case, you specify the transaction attributes in the bean's deployment descriptor. You do not have to code the transaction boundaries in the bean; the container marks the boundaries for you. See Chapter 30 for more information.

Primary Key

Each entity bean has a unique object identifier. A customer entity bean, for example, might be identified by a customer number. The unique identifier, or *primary key*, enables the client to locate a particular entity bean. For more information, see the section Primary Keys for Bean-Managed Persistence (page 204).

Relationships

Like a table in a relational database, an entity bean may be related to other entity beans. For example, in a college enrollment application, StudentBean and CourseBean would be related because students enroll in classes.

You implement relationships differently for entity beans with bean-managed persistence than those with container-managed persistence. With bean-managed persistence, the code that you write implements the relationships. But with container-managed persistence, the EJB container takes care of the relationships for you. For this reason, relationships in entity beans with container-managed persistence are often referred to as *container-managed relationships*.

Container-Managed Persistence

The term container-managed persistence means that the EJB container handles all database access required by the entity bean. The bean's code contains no database access (SQL) calls. As a result, the bean's code is not tied to a specific persistent storage mechanism (database). Because of this flexibility, even if you redeploy the same entity bean on different J2EE servers that use different databases, you won't need to modify or recompile the bean's code. In short, your entity beans are more portable if you use container-managed persistence than if they use bean-managed persistence.

To generate the data access calls, the container needs information that you provide in the entity bean's abstract schema.

Abstract Schema

Part of an entity bean's deployment descriptor, the *abstract schema* defines the bean's persistent fields and relationships. The term *abstract* distinguishes this schema from the physical schema of the underlying data store. In a relational database, for example, the physical schema is made up of structures such as tables and columns.

You specify the name of an abstract schema in the deployment descriptor. This name is referenced by queries written in the Enterprise JavaBeans Query Language (EJB QL). For an entity bean with container-managed persistence, you must define an EJB QL query for every finder method (except findByPrimaryKey). The EJB QL query determines the query that is executed by the EJB container when the finder method is invoked. To learn more about EJB QL, see Chapter 29.

You'll probably find it helpful to sketch the abstract schema before writing any code. Figure 4–1 represents a simple abstract schema that describes the relationships between three entity beans. These relationships are discussed further in the sections that follow.

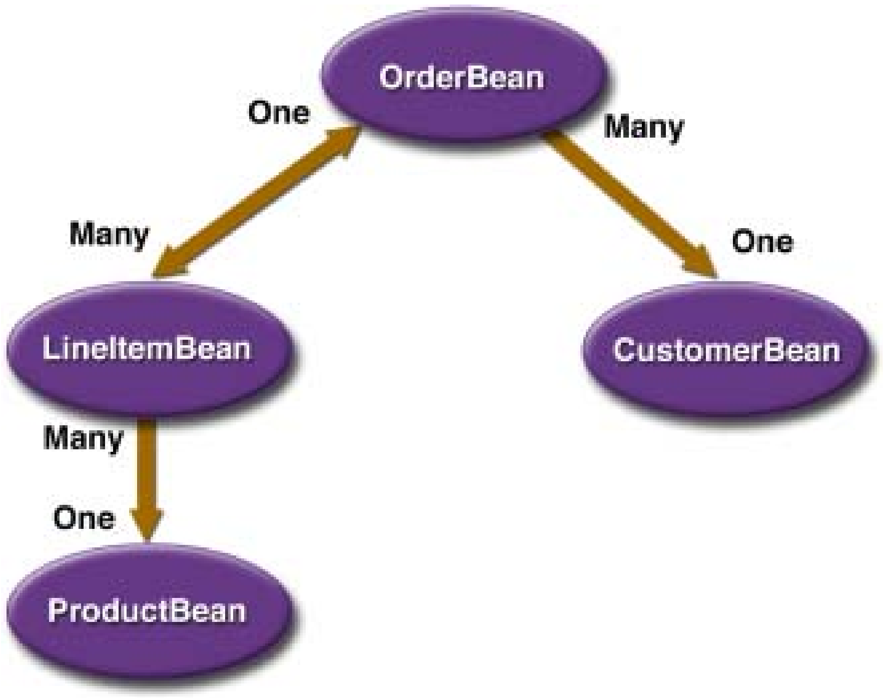


Figure 4–1 A High-Level View of an Abstract Schema

Persistent Fields

The persistent fields of an entity bean are stored in the underlying data store. Collectively, these fields constitute the state of the bean. At runtime, the EJB container automatically synchronizes this state with the database. During deployment, the container typically maps the entity bean to a database table and maps the persistent fields to the table's columns.

A CustomerBean entity bean, for example, might have persistent fields such as firstName, lastName, phone, and emailAddress. In container-managed persistence, these fields are virtual. You declare them in the abstract schema, but you do not code them as instance variables in the entity bean class. Instead, the persistent fields are identified in the code by access methods (getters and setters).

Relationship Fields

A *relationship field* is like a foreign key in a database table: it identifies a related bean. Like a persistent field, a relationship field is virtual and is defined in the enterprise bean class via access methods. But unlike a persistent field, a relationship field does not represent the bean's state. Relationship fields are discussed further in Direction in Container-Managed Relationships (page 113).

Multiplicity in Container-Managed Relationships

There are four types of multiplicities: one-to-one, one-to-many, many-to-one, and many-to-many.

One-to-one: Each entity bean instance is related to a single instance of another entity bean. For example, to model a physical warehouse in which each storage bin contains a single widget, StorageBinBean and WidgetBean would have a one-to-one relationship.

One-to-many: An entity bean instance can be related to multiple instances of the other entity bean. A sales order, for example, can have multiple line items. In the order application, OrderBean would have a one-to-many relationship with LineItemBean.

Many-to-one: Multiple instances of an entity bean can be related to a single instance of the other entity bean. This multiplicity is the opposite of a one-to-many relationship. In the example just mentioned, from the perspective of LineItemBean the relationship to OrderBean is many-to-one.

Many-to-many: The entity bean instances can be related to multiple instances of each other. For example, in college each course has many students, and every student may take several courses. Therefore, in an enrollment application, CourseBean and StudentBean would have a many-to-many relationship.

Direction in Container-Managed Relationships

The direction of a relationship can be either bidirectional or unidirectional. In a *bidirectional* relationship, each entity bean has a relationship field that refers to the other bean. Through the relationship field, an entity bean's code can access its related object. If an entity bean has a relative field, then we often say that it "knows" about its related object. For example, if OrderBean knows what LineItemBean instances it has and if LineItemBean knows what OrderBean it belongs to, then they have a bidirectional relationship.

In a *unidirectional* relationship, only one entity bean has a relationship field that refers to the other. For example, LineItemBean would have a relationship field that identifies ProductBean, but ProductBean would not have a relationship field for LineItemBean. In other words, LineItemBean knows about ProductBean, but ProductBean doesn't know which LineItemBean instances refer to it.

EJB QL queries often navigate across relationships. The direction of a relationship determines whether a query can navigate from one bean to another. For example, a query can navigate from LineItemBean to ProductBean but cannot navigate in the opposite direction. For OrderBean and LineItemBean, a query could navigate in both directions, because these two beans have a bidirectional relationship.

When to Use Entity Beans

You should probably use an entity bean under the following conditions:

- The bean represents a business entity and not a procedure. For example, CreditCardBean would be an entity bean, but CreditCardVerifierBean would be a session bean.
- The bean's state must be persistent. If the bean instance terminates or if the Application Server is shut down, the bean's state still exists in persistent storage (a database).

What Is a Message-Driven Bean?

A *message-driven bean* is an enterprise bean that allows J2EE applications to process messages asynchronously. It normally acts as a JMS message listener, which is similar to an event listener except that it receives JMS messages instead of events. The messages can be sent by any J2EE component—an application client, another enterprise bean, or a web component—or by a JMS application or system that does not use J2EE technology. Message-driven beans can process either JMS messages or other kinds of messages.

For a simple code sample, see Chapter 9. For more information about using message-driven beans, see Using the JMS API in a J2EE Application (page 1248) and Chapter 34.

What Makes Message-Driven Beans Different from Session and Entity Beans?

The most visible difference between message-driven beans and session and entity beans is that clients do not access message-driven beans through interfaces. Interfaces are described in the section Defining Client Access with Interfaces (page 116). Unlike a session or entity bean, a message-driven bean has only a bean class.

In several respects, a message-driven bean resembles a stateless session bean.

- A message-driven bean's instances retain no data or conversational state for a specific client.
- All instances of a message-driven bean are equivalent, allowing the EJB container to assign a message to any message-driven bean instance. The container can pool these instances to allow streams of messages to be processed concurrently.
- A single message-driven bean can process messages from multiple clients.

The instance variables of the message-driven bean instance can contain some state across the handling of client messages—for example, a JMS API connection, an open database connection, or an object reference to an enterprise bean object.

Client components do not locate message-driven beans and invoke methods directly on them. Instead, a client accesses a message-driven bean through JMS by sending messages to the message destination for which the message-driven bean class is the MessageListener. You assign a message-driven bean's destination during deployment by using Application Server resources.

Message-driven beans have the following characteristics:

- They execute upon receipt of a single client message.
- They are invoked asynchronously.
- They are relatively short-lived.
- They do not represent directly shared data in the database, but they can access and update this data.
- They can be transaction-aware.
- They are stateless.

When a message arrives, the container calls the message-driven bean's onMessage method to process the message. The onMessage method normally casts the message to one of the five JMS message types and handles it in accordance with the application's business logic. The onMessage method can call helper methods, or it can invoke a session or entity bean to process the information in the message or to store it in a database.

A message can be delivered to a message-driven bean within a transaction context, so all operations within the onMessage method are part of a single transaction. If message processing is rolled back, the message will be redelivered. For more information, see Chapter 9.

When to Use Message-Driven Beans

Session beans and entity beans allow you to send JMS messages and to receive them synchronously, but not asynchronously. To avoid tying up server resources, you may prefer not to use blocking synchronous receives in a server-side component. To receive messages asynchronously, use a message-driven bean.

Defining Client Access with Interfaces

The material in this section applies only to session and entity beans and not to message-driven beans. Because they have a different programming model, message-driven beans do not have interfaces that define client access.

A client can access a session or an entity bean only through the methods defined in the bean's interfaces. These interfaces define the client's view of a bean. All other aspects of the bean—method implementations, deployment descriptor settings, abstract schemas, and database access calls—are hidden from the client.

Well-designed interfaces simplify the development and maintenance of J2EE applications. Not only do clean interfaces shield the clients from any complexities in the EJB tier, but they also allow the beans to change internally without affecting the clients. For example, even if you change your entity beans from bean-managed to container-managed persistence, you won't have to alter the client code. But if you were to change the method definitions in the interfaces, then you might have to modify the client code as well. Therefore, to isolate your clients from possible changes in the beans, it is important that you design the interfaces carefully.

When you design a J2EE application, one of the first decisions you make is the type of client access allowed by the enterprise beans: remote, local, or web service.

Remote Clients

A remote client of an enterprise bean has the following traits:

- It can run on a different machine and a different Java virtual machine (JVM) than the enterprise bean it accesses. (It is not required to run on a different JVM.)
- It can be a web component, an application client, or another enterprise bean.
- To a remote client, the location of the enterprise bean is transparent.

To create an enterprise bean that has remote access, you must code a remote interface and a home interface. The *remote interface* defines the business methods that are specific to the bean. For example, the remote interface of a bean named BankAccountBean might have business methods named deposit and credit. The *home interface* defines the bean's life-cycle methods: create and remove. For entity beans, the home interface also defines finder methods and home methods. *Finder methods* are used to locate entity beans. *Home methods* are business methods that are invoked on all instances of an entity bean class. Figure 4–2 shows how the interfaces control the client's view of an enterprise bean.

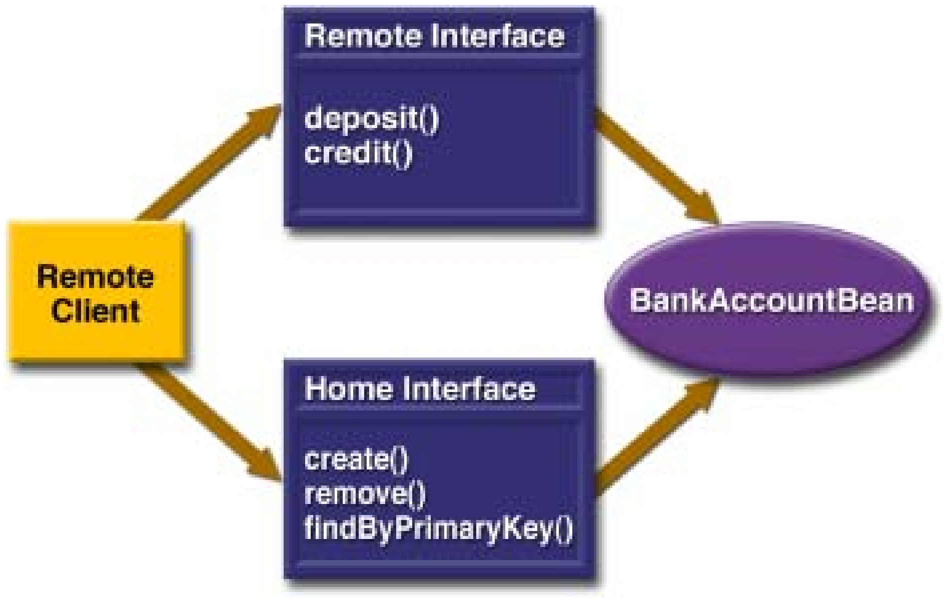


Figure 4–2 Interfaces for an Enterprise Bean with Remote Access

Local Clients

A local client has these characteristics:

- It must run in the same JVM as the enterprise bean it accesses.
- It can be a web component or another enterprise bean.
- To the local client, the location of the enterprise bean it accesses is not transparent.
- It is often an entity bean that has a container-managed relationship with another entity bean.

To build an enterprise bean that allows local access, you must code the local interface and the local home interface. The *local interface* defines the bean's business methods, and the *local home* interface defines its life-cycle and finder methods.

Local Interfaces and Container-Managed Relationships

If an entity bean is the target of a container-managed relationship, then it must have local interfaces. The direction of the relationship determines whether or not a bean is the target. In Figure 4–1, for example, ProductBean is the target of a unidirectional relationship with LineItemBean. Because LineItemBean accesses ProductBean locally, ProductBean must have the local interfaces. LineItemBean also needs local interfaces, not because of its relationship with ProductBean, but because it is the target of a relationship with OrderBean. And because the relationship between LineItemBean and OrderBean is bidirectional, both beans must have local interfaces.

Because they require local access, entity beans that participate in a container-managed relationship must reside in the same EJB JAR file. The primary benefit of this locality is increased performance: local calls are usually faster than remote calls.

Deciding on Remote or Local Access

Whether to allow local or remote access depends on the following factors.

- *Container-managed relationships*: If an entity bean is the target of a container-managed relationship, it must use local access.
- *Tight or loose coupling of related beans*: Tightly coupled beans depend on one another. For example, a completed sales order must have one or more line items, which cannot exist without the order to which they belong. The OrderBean and LineItemBean entity beans that model this relationship are tightly coupled. Tightly coupled beans are good candidates for local access. Because they fit together as a logical unit, they probably call each other often and would benefit from the increased performance that is possible with local access.
- *Type of client*: If an enterprise bean is accessed by application clients, then it should allow remote access. In a production environment, these clients almost always run on different machines than the Application Server does. If an enterprise bean's clients are web components or other enterprise beans, then the type of access depends on how you want to distribute your components.
- *Component distribution*: J2EE applications are scalable because their server-side components can be distributed across multiple machines. In a distributed application, for example, the web components may run on a different server than do the enterprise beans they access. In this distributed scenario, the enterprise beans should allow remote access.
- *Performance*: Because of factors such as network latency, remote calls may be slower than local calls. On the other hand, if you distribute components among different servers, you might improve the application's overall performance. Both of these statements are generalizations; actual performance can vary in different operational environments. Nevertheless, you should keep in mind how your application design might affect performance.

If you aren't sure which type of access an enterprise bean should have, then choose remote access. This decision gives you more flexibility. In the future you can distribute your components to accommodate growing demands on your application.

Although it is uncommon, it is possible for an enterprise bean to allow both remote and local access. Such a bean would require both remote and local interfaces.

Web Service Clients

A web service client can access a J2EE application in two ways. First, the client can access a web service created with JAX-RPC. (For more information on JAX-RPC, see Chapter 2, Building Web Services with JAX-RPC, page 29.) Second, a web service client can invoke the business methods of a stateless session bean. Other types of enterprise beans cannot be accessed by web service clients.

Provided that it uses the correct protocols (SOAP, HTTP, WSDL), any web service client can access a stateless session bean, whether or not the client is written in the Java programming language. The client doesn't even "know" what technology implements the service—stateless session bean, JAX-RPC, or some other technology. In addition, enterprise beans and web components can be clients of web services. This flexibility enables you to integrate J2EE applications with web services.

A web service client accesses a stateless session bean through the bean's web service endpoint interface. Like a remote interface, a *web service endpoint interface* defines the business methods of the bean. In contrast to a remote interface, a web service endpoint interface is not accompanied by a home interface, which defines the bean's life-cycle methods. The only methods of the bean that may be invoked by a web service client are the business methods that are defined in the web service endpoint interface.

For a code sample, see The HelloService Web Service Example (page 153).

Method Parameters and Access

The type of access affects the parameters of the bean methods that are called by clients. The following topics apply not only to method parameters but also to method return values.

Isolation

The parameters of remote calls are more isolated than those of local calls. With remote calls, the client and bean operate on different copies of a parameter object. If the client changes the value of the object, the value of the copy in the bean does not change. This layer of isolation can help protect the bean if the client accidentally modifies the data.

In a local call, both the client and the bean can modify the same parameter object. In general, you should not rely on this side effect of local calls. Perhaps someday you will want to distribute your components, replacing the local calls with remote ones.

As with remote clients, web service clients operate on different copies of parameters than does the bean that implements the web service.

Granularity of Accessed Data

Because remote calls are likely to be slower than local calls, the parameters in remote methods should be relatively coarse-grained. A coarse-grained object contains more data than a fine-grained one, so fewer access calls are required. For the same reason, the parameters of the methods called by web service clients should also be coarse-grained.

For example, suppose that a CustomerBean entity bean is accessed remotely. This bean would have a single getter method that returns a CustomerDetails object, which encapsulates all of the customer's information. But if CustomerBean is to be accessed locally, it could have a getter method for each instance variable: getFirstName, getLastName, getPhoneNumber, and so forth. Because local calls are fast, the multiple calls to these finer-grained getter methods would not significantly degrade performance.

The Contents of an Enterprise Bean

To develop an enterprise bean, you must provide the following files:

- *Deployment descriptor*: An XML file that specifies information about the bean such as its persistence type and transaction attributes. The deploytool utility creates the deployment descriptor when you step through the New Enterprise Bean wizard.
- *Enterprise bean class*: Implements the methods defined in the following interfaces.
- *Interfaces*: The remote and home interfaces are required for remote access. For local access, the local and local home interfaces are required. For access by web service clients, the web service endpoint interface is required. See the section Defining Client Access with Interfaces (page 116). (Please note that these interfaces are not used by message-driven beans.)

- *Helper classes*: Other classes needed by the enterprise bean class, such as exception and utility classes.

You package the files in the preceding list into an EJB JAR file, the module that stores the enterprise bean. An EJB JAR file is portable and can be used for different applications. To assemble a J2EE application, you package one or more modules—such as EJB JAR files—into an EAR file, the archive file that holds the application. When you deploy the EAR file that contains the bean's EJB JAR file, you also deploy the enterprise bean onto the Application Server. You can also deploy an EJB JAR that is not contained in an EAR file.

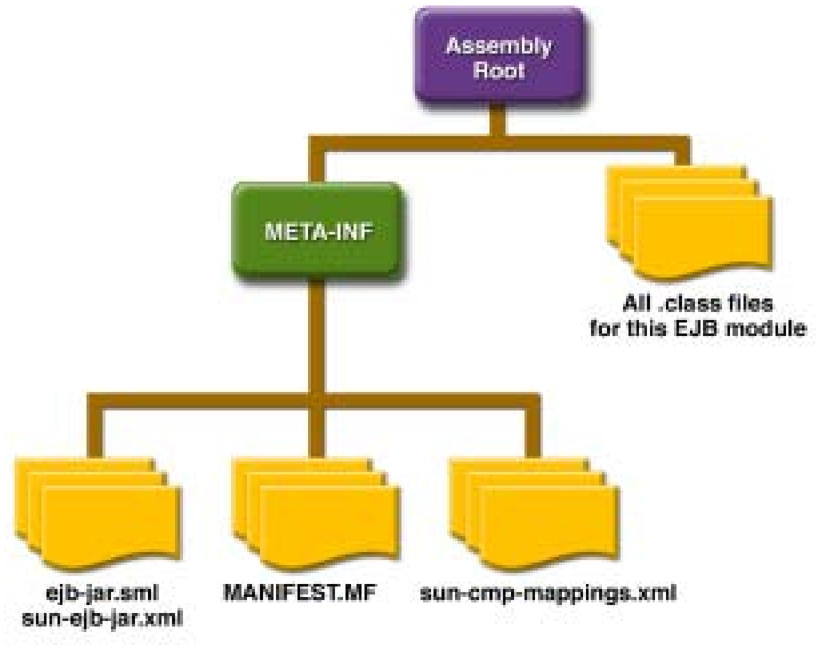


Figure 4–3 Structure of an Enterprise Bean JAR

Naming Conventions for Enterprise Beans

Because enterprise beans are composed of multiple parts, it's useful to follow a naming convention for your applications. Table 4–2 summarizes the conventions for the example beans in this tutorial.

Table 4–2 Naming Conventions for Enterprise Beans

Item	Syntax	Example
Enterprise bean name (DD[a])	<*name*>Bean	AccountBean
EJB JAR display name (DD)	<*name*>JAR	AccountJAR
Enterprise bean class	<*name*>Bean	AccountBean
Home interface	<*name*>Home	AccountHome
Remote interface	<*name*>	Account
Local home interface	<*name*>LocalHome	AccountLocalHome
Local interface	<*name*>Local	AccountLocal
Abstract schema (DD)	<*name*>	Account

a.*DD* means that the item is an element in the bean's deployment descriptor.

The Life Cycles of Enterprise Beans

An enterprise bean goes through various stages during its lifetime, or life cycle. Each type of enterprise bean—session, entity, or message-driven—has a different life cycle.

The descriptions that follow refer to methods that are explained along with the code examples in the next two chapters. If you are new to enterprise beans, you should skip this section and try out the code examples first.

The Life Cycle of a Stateful Session Bean

Figure 4–4 illustrates the stages that a session bean passes through during its lifetime. The client initiates the life cycle by invoking the create method. The EJB container instantiates the bean and then invokes the setSessionContext and ejbCreate methods in the session bean. The bean is now ready to have its business methods invoked.

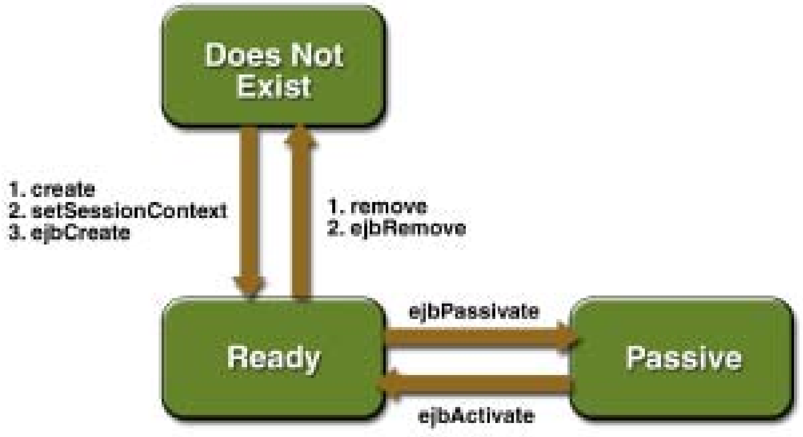


Figure 4–4 Life Cycle of a Stateful Session Bean

While in the ready stage, the EJB container may decide to deactivate, or *passivate*, the bean by moving it from memory to secondary storage. (Typically, the EJB container uses a least-recently-used algorithm to select a bean for passivation.) The EJB container invokes the bean's ejbPassivate method immediately before passivating it. If a client invokes a business method on the bean while it is in the passive stage, the EJB container activates the bean, calls the bean's ejbActivate method, and then moves it to the ready stage.

At the end of the life cycle, the client invokes the remove method, and the EJB container calls the bean's ejbRemove method. The bean's instance is ready for garbage collection.

Your code controls the invocation of only two life-cycle methods: the create and remove methods in the client. All other methods in Figure 4–4 are invoked by the EJB container. The ejbCreate method, for example, is inside the bean class, allowing you to perform certain operations right after the bean is instantiated. For example, you might wish to connect to a database in the ejbCreate method. See Chapter 31 for more information.

The Life Cycle of a Stateless Session Bean

Because a stateless session bean is never passivated, its life cycle has only two stages: nonexistent and ready for the invocation of business methods. Figure 4–5 illustrates the stages of a stateless session bean.

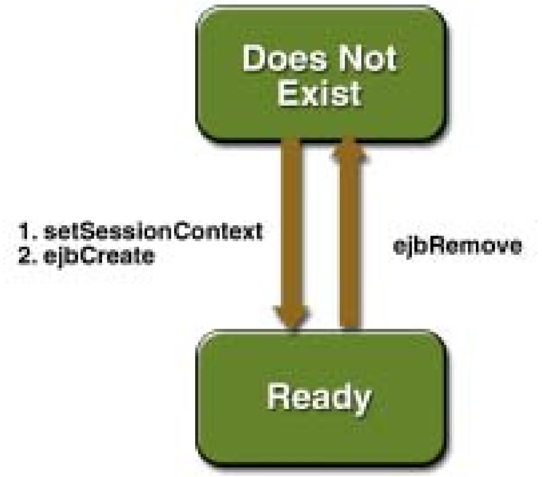


Figure 4–5 Life Cycle of a Stateless Session Bean

The Life Cycle of an Entity Bean

Figure 4–6 shows the stages that an entity bean passes through during its lifetime. After the EJB container creates the instance, it calls the setEntityContext method of the entity bean class. The setEntityContext method passes the entity context to the bean.

After instantiation, the entity bean moves to a pool of available instances. While in the pooled stage, the instance is not associated with any particular EJB object identity. All instances in the pool are identical. The EJB container assigns an identity to an instance when moving it to the ready stage.

There are two paths from the pooled stage to the ready stage. On the first path, the client invokes the create method, causing the EJB container to call the ejbCreate and ejbPostCreate methods. On the second path, the EJB container invokes the ejbActivate method. While an entity bean is in the ready stage, an it's business methods can be invoked.

There are also two paths from the ready stage to the pooled stage. First, a client can invoke the remove method, which causes the EJB container to call the ejbRemove method. Second, the EJB container can invoke the ejbPassivate method.

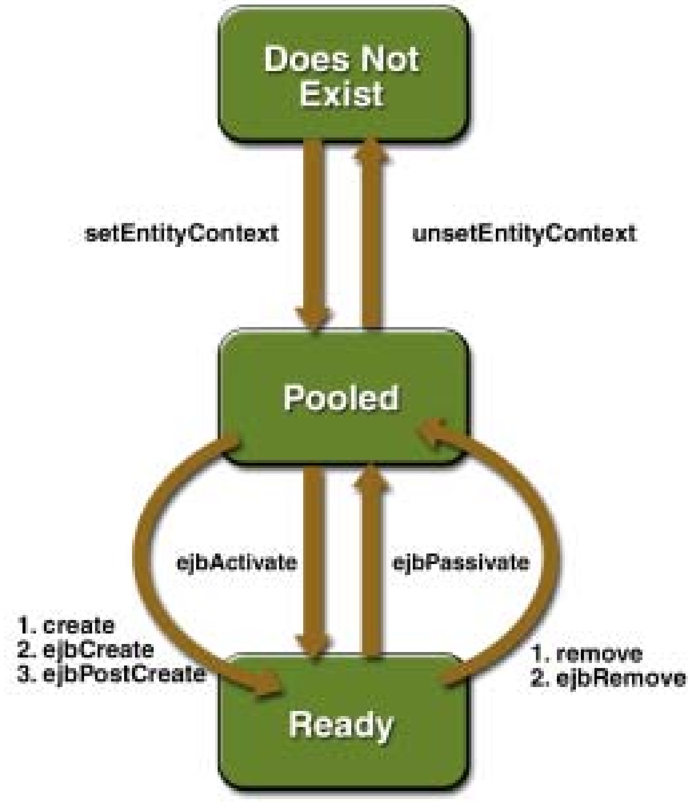


Figure 4–6 Life Cycle of an Entity Bean

At the end of the life cycle, the EJB container removes the instance from the pool and invokes the unsetEntityContext method.

In the pooled state, an instance is not associated with any particular EJB object identity. With bean-managed persistence, when the EJB container moves an instance from the pooled state to the ready state, it does not automatically set the primary key. Therefore, the ejbCreate and ejbActivate methods must assign a value to the primary key. If the primary key is incorrect, the ejbLoad and ejbStore methods cannot synchronize the instance variables with the database. In the section The SavingsAccountBean Example (page 167), the ejbCreate method assigns the primary key from one of the input parameters. The ejbActivate method sets the primary key (id) as follows:

```
id = (String)context.getPrimaryKey();
```

In the pooled state, the values of the instance variables are not needed. You can make these instance variables eligible for garbage collection by setting them to null in the ejbPassivate method.

The Life Cycle of a Message-Driven Bean

Figure 4–7 illustrates the stages in the life cycle of a message-driven bean.

The EJB container usually creates a pool of message-driven bean instances. For each instance, the EJB container instantiates the bean and performs these tasks:

1. It calls the setMessageDrivenContext method to pass the context object to the instance.
2. It calls the instance's ejbCreate method.

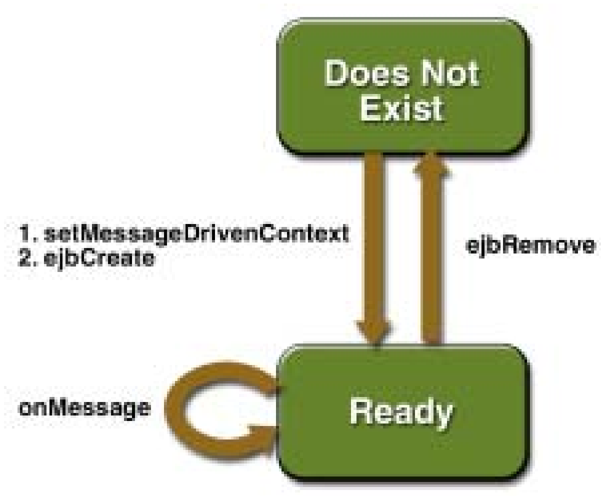


Figure 4–7 Life Cycle of a Message-Driven Bean

Like a stateless session bean, a message-driven bean is never passivated, and it has only two states: nonexistent and ready to receive messages.

At the end of the life cycle, the container calls the ejbRemove method. The bean's instance is then ready for garbage collection.

Further Information

For further information on Enterprise JavaBeans technology, see the following:

- Enterprise JavaBeans 2.1 specification:
  http://java.sun.com/products/ejb/docs.html
- The Enterprise JavaBeans web site:
  http://java.sun.com/products/ejb

5

Getting Started with Enterprise Beans

THIS chapter shows how to develop, deploy, and run a simple J2EE application named `ConverterApp`. The purpose of `ConverterApp` is to calculate currency conversions between yen and eurodollars. `ConverterApp` consists of an enterprise bean, which performs the calculations, and a web client.

Here's an overview of the steps you'll follow in this chapter:

1. Create the J2EE application: `ConverterApp`.
2. Create the enterprise bean: `ConverterBean`.
3. Create the web client in `ConverterWAR`.
4. Deploy `ConverterApp` onto the server.
5. Using a browser, run the web client.

Before proceeding, make sure that you've done the following:

- Read Chapter 1.
- Become familiar with enterprise beans (see Chapter 4).

Creating the J2EE Application

In this section, you'll create a project named `ConverterApp` to store the J2EE application.

1. In the IDE, choose File→New Project (Ctrl-Shift-N).
2. From the Enterprise category, select Enterprise Application and click Next.
3. Name the project `ConverterApp`, specify a location for the project and click Finish.

This wizard actually creates three projects: one for the enterprise application, one for the EJB module, and one for the web module.

Creating the Enterprise Bean

The enterprise bean in our example is a stateless session bean called `ConverterBean`. The source code for `ConverterBean` is in the <*INSTALL*>`/j2eetutorial14/examples/ejb/converter/ConverterApp/ConverterApp-ejb/` directory.

Creating `ConverterBean` requires these steps:

1. Generating the bean classes and interfaces from a NetBeans template
2. Adding business methods to the enterprise bean.

Creating the ConverterBean Enterprise Bean

The enterprise bean templates automatically create all of the classes and interfaces necessary for the enterprise bean and register the enterprise bean in the EJB module's deployment descriptor.

1. In the Projects window, right-click the ConverterApp-EJBModule node and choose New→Session Bean.
2. In the EJB Name field, type `Converter`. In the Package field, type `converter`. Set the bean to be stateless and remote and click Finish.

The IDE creates the following classes:

- `ConverterBean.java`. The enterprise bean class. All of the EJB infrastructure methods are generated automatically and are hidden in a code fold.
- `ConverterRemote.java`. The *remote interface*. The remote interface usually defines the business methods that a client can call. The business methods are implemented in the enterprise bean code. Because the IDE enforces best coding practices, it actually registers all of the business methods in a remote business interface, which the remote interface extends.
- `ConverterRemoteBusiness.java`. The *business interface*. Presently this class is empty, but as we add business methods to the bean this class will be populated.
- `ConverterRemoteHome.java`. The *home interface*. A home interface defines the methods that allow a client to create, find, or remove an enterprise bean.

Adding Business Methods

1. Expand the Enterprise JavaBeans node, right-click the ConverterSB node, and choose Add→Business Method.
2. In the dialog box, type `dollarToYen` in the Name field and `BigDecimal` in the Return Type field. In the Parameters tab, click Add to add a `BigDecimal` parameter named `dollars`. Then click OK to add the business method.
3. Repeat steps 1 and 2 to add a business method called `yenToEuro` that returns a `BigDecimal` and has one `BigDecimal` parameter named `yen`.
4. Press Alt-Shift-F to generate an import statement for `java.math.bigDecimal`.
5. Expand the Source Packages node and the `converter` package node. Double-click `ConverterRemoteBusiness.java` to open it in the Source Editor. Notice that the IDE has automatically declared the `dollarToYen` and `yenToEuro` methods in the interface. Press Alt-Shift-F to generate an import statement for `java.math.bigDecimal`.
6. In `ConverterBean.java`, add the following field declarations right below the class declaration:

   ```
   BigDecimal yenRate = new BigDecimal("121.6000");
   BigDecimal euroRate = new BigDecimal("0.0077");
   ```

7. In `ConverterBean.java`, implement the `dollarToYen` and `yenToEuro` methods as follows:

```
  public BigDecimal dollarToYen(BigDecimal dollars) {
    BigDecimal result = dollars.multiply(yenRate);
    return result.setScale(2, BigDecimal.ROUND_UP);
  }

  public BigDecimal yenToEuro(BigDecimal yen) {
    BigDecimal result = yen.multiply(euroRate);
    return result.setScale(2, BigDecimal.ROUND_UP);
  }
```

The full source code for the `ConverterBean` class follows.

```
import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import java.math.*;

public class ConverterBean implements SessionBean {

  private SessionContext context;
  BigDecimal yenRate = new BigDecimal("121.6000");
  BigDecimal euroRate = new BigDecimal("0.0077");

  public void setSessionContext(SessionContext aContext) {
    context = aContext;
  }
  public void ejbRemove() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void ejbCreate() {}

  public BigDecimal dollarToYen(BigDecimal dollars) {
    BigDecimal result = dollars.multiply(yenRate);
    return result.setScale(2,BigDecimal.ROUND_UP);
  }

  public BigDecimal yenToEuro(BigDecimal yen) {
    BigDecimal result = yen.multiply(euroRate);
    return result.setScale(2,BigDecimal.ROUND_UP);
   }

}
```

Creating the Web Client

The web client is contained in the *<INSTALL>*`/j2eetutorial14/examples/ejb/converter/ConverterApp/ConverterApp-war/` directory. The web client is implemented in a servlet, `ConverterServlet.java`.

Coding the Web Client

When you access an enterprise bean from a servlet, the IDE provides tools that do much of the work for you. For instance, the IDE automatically generates lookup code for the bean and adds the bean reference to the web module's deployment descriptors.

1. In the Projects window, right-click the ConverterApp-WebModule node and choose New→Servlet.
2. Name the servlet `ConverterServlet` and place it in a package called `converter`. Click Next.
3. Leave the default settings for all of the information in the last page of the wizard and click Finish.

Locating the Home Interface

1. In the Source Editor, right-click anywhere in the body of the `ConverterServlet` class and choose Enterprise Resources→Call Enterprise Bean.
2. In the dialog box, select ConverterSB and click OK. The IDE generates the `lookupConverterBean` method at the bottom of the file.

The IDE adds the `lookupConverterBean` method to the servlet and registers the bean reference in the web module's deployment descriptors. The lookup code does the following:

1. Create an initial naming context.

   ```
   javax.naming.Context c = new javax.naming.InitialContext();
   ```

   The `Context` interface is part of the Java Naming and Directory Interface (JNDI). A *naming context* is a set of name-to-object bindings. A name that is bound within a context is the *JNDI name* of the object.

   An `InitialContext` object, which implements the `Context` interface, provides the starting point for the resolution of names. All naming operations are relative to a context.

2. Obtain the environment naming context of the web client and retrieves the object bound to the name `ejb/ConverterBean`.

   ```
   Object remote = c.lookup("java:comp/env/ejb/ConverterBean");
   ```

   The `java:comp/env` name is bound to the environment naming context of the `ConverterApp-WebModule` component.

   The `ejb/ConverterBean` name is bound to an *enterprise bean reference*, a logical name for the home of an enterprise bean. In this case, the `ejb/ConverterBean` name refers to the `ConverterRemoteHome` object. The names of enterprise beans should reside in the `java:comp/env/ejb` subcontext.

3. Narrow the reference to a `ConverterRemoteHome` object.

   ```
   converter.ConverterRemoteHome rv =
        (converter.ConverterRemoteHome)
        javax.rmi.PortableRemoteObject.narrow(remote,
          converter.ConverterRemoteHome.class);
   ```

4. Creates an instance of the `ConverterBean` enterprise bean:

   ```
   return rv.create();
   ```

Invoking Business Methods

1. In the Source Editor, go to the `processRequest` method and remove the comment symbols that comment out the text between `PrintWriter out = response.getWriter();` and `out.close();`. (You can put the insertion point on each line and press Ctrl-E to delete the entire line.)

2. Add the following code in the body of the servlet, between `out.println("<body>");` and `out.println("</body>");`:

   ```
   out.println("<h1><b><center>Converter</center></b></h1>");
   out.println("<hr>");
   out.println("<p>Enter an amount to convert:</p>");
   out.println("<form method=\"get\">");
   out.println("<input type=\"text\"
        name=\"amount\" size=\"25\">");
   out.println("<br>");
   out.println("<p>");
   out.println("<input type=\"submit\" value=\"Submit\">");
   out.println("<input type=\"reset\" value=\"Reset\">");
   out.println("</form>");
   String amount = request.getParameter("amount");
   if ( amount != null && amount.length() > 0 ) {
   ```

```
        try {
            converter.ConverterRemote converter;
            converter = lookupConverterBean();

            java.math.BigDecimal d =
                    new java.math.BigDecimal(amount);
            out.println("<p>");
            out.println("<p>");
            out.println(amount + " Dollars are  "
                    + converter.dollarToYen(d) + " Yen.");
            out.println("<p>");
            out.println(amount + " Yen are "
                    + converter.yenToEuro(d) + " Euro.");

            converter.remove();
        } catch (Exception e){
            out.println("Cannot lookup or execute EJB!");
        }

    }
```

Specifying the Enterprise Application's Default URL

By default, the IDE opens the web module's index.jsp page when you run the enterprise application. You need to change this setting to open `ConverterServlet.java` instead.

1. In the Projects window, right-click the ConverterApp project node and choose Properties.
2. Click Run in the left pane of the dialog box.
3. Type `/ConverterServlet` in the Relative URL field and click OK.

Deploying the J2EE Application

Now that the J2EE application contains the components, it is ready for deployment.

1. In the Projects window, right-click the ConverterApp node and choose Run Project.

The IDE does all of the following:

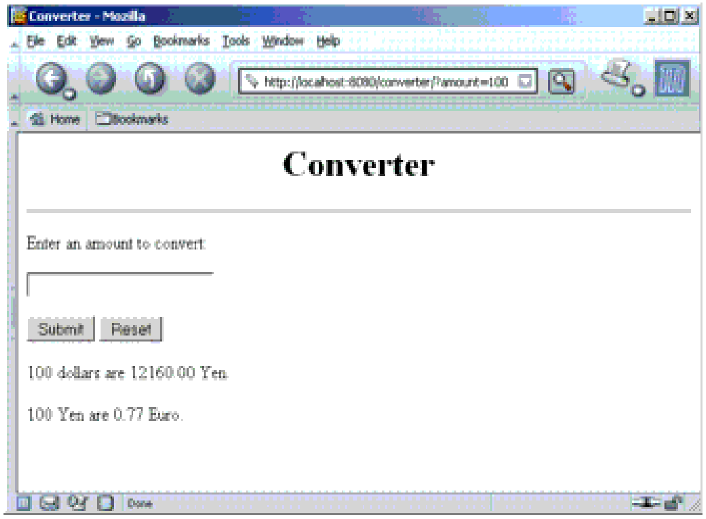
- Starts the application server if it is not already started.
- Builds the ConverterApp project and the projects for each of its modules. You can view the build ouputs in the Files window.
- Deploys converterapp.ear to the application server.
- Opens your default web browser at the following URL: `http://<host>:<port>/ConverterApp-WebModule/ConverterServlet`

Running the Web Client

As stated above, the IDE automatically runs the web client every time you run the ConverterApp project. Once the enterprise application is deployed to a running application server, you can access the application client at any time by pointing your browser at the following URL. Replace `<host>` with the name of the host running the Application Server. If your browser is running on the same host as the Application Server, you can replace `<host>` with `localhost`.

```
http://<host>:<port>/ConverterApp-WebModule/ConverterServlet
```

After entering `100` in the input field and clicking Submit, you should see the screen shown in Figure 5–1.


Converter - Mozilla
File Edit View Go Bookmarks Tools Window Help
http://localhost:8080/converter/?amount=100
Home
Bookmarks
Submit
Reset
Done

Converter

Enter an amount to convert:

100 dollars are 12160.00 Yen.

100 Yen are 0.77 Euro.

Figure 5–1 `ConverterApp` Web Client

Modifying the J2EE Application

The Application Server and the NetBeans IDE support iterative development. Whenever you make a change to a J2EE application, you must redeploy the application.

Modifying a Deployment Setting

To modify a deployment setting of `ConverterApp`, you edit the appropriate field in the deployment descriptors and redeploy. For example, to change a JNDI name from `ATypo` to `ConverterBean`, you would follow these steps.

1. In the Projects window, expand the Configuration Files node for the ConverterApp-EJBModule project and double-click `sun-ejb-jar.xml`.
2. In the left pane of the deployment descriptor editor, expand the Sun Configuration node and select ConverterBean [EJB].
3. In the JNDI Name field, enter `ejb/ConverterBean`.

4. In the Projects window, right-click the ConverterApp project and choose Run Project. The IDE saves all the files, rebuilds the project, and redeploys it to the application server.

6
Session Bean Examples

SESSION beans are powerful because they extend the reach of your clients into remote servers. In Chapter 5, you built a stateless session bean named ConverterBean. This chapter examines the source code of three more session beans:

- CartBean: a stateful session bean that is accessed by a remote client
- HelloServiceBean: a stateless session bean that implements a web service
- TimerSessionBean: a stateless session bean that sets a timer

The CartBean Example Application

The CartBean session bean represents a shopping cart in an online bookstore. The bean's client can add a book to the cart, remove a book, or retrieve the cart's contents. To construct the CartBean example, you need to create the following components:

- Cart EJB module
- CartClient application

The Cart EJB module contains the session bean and the interfaces. When creating the session bean, the IDE creates the following components:

- Session bean class (CartBean)
- Remote interface (CartRemote)
- Home interface (CartRemoteHome)
- Business interface (CartRemoteBusiness)

All session beans require a session bean class. All enterprise beans that permit remote access must have a home and a remote interface.

To meet the needs of a specific application, an enterprise bean may also need some helper classes. The `CartBean` session bean uses two helper classes (`BookException` and `IdVerifier`) which are discussed in the section Helper Classes.

The source code for this example is located in the <*INSTALL*>`/j2eetutorial14/examples/ejb/cart/` directory.

Creating the Cart EJB Project

In the IDE, you first need to create a project for the EJB module for the session bean.

1. Choose File→New Project (Ctrl-Shift-N).
2. In the New Project wizard, choose the Enterprise template category, select EJB Module in the Projects pane and click Next.
3. Type `Cart` as the Project Name, specify a Project Location and click Finish.

The Cart module appears in the Projects window of the IDE. The next step is to add a session bean to the module.

Creating the Session Bean

For this example, you create a session bean class called `CartBean`. Like any session bean, the `CartBean` class must meet these requirements:

- It implements the `SessionBean` interface.
- The class is defined as `public`.
- The class cannot be defined as `abstract` or `final`.
- It implements one or more `ejbCreate` methods.
- It implements the business methods.
- It contains a `public` constructor with no parameters.
- It must not define the `finalize` method.

When you create the session bean in the IDE, the required infrastructure methods and session bean interfaces are generated automatically.

1. Right-click the Cart node in the Projects window and choose New→Session Bean.
2. Type `Cart` as the EJB Name, `cart` as the Package name, and select Stateful and Remote as the session and interface types (deselect the Local interface type). Then click Finish.

The IDE creates the Cart session bean under the Enterprise Beans node and opens the `CartBean` class in the Source Editor. You can see that the `CartBean` class automatically implements the remote interfaces and also creates the required session bean infrastructure methods.

The Cart session bean interface declares the `ejbRemove`, `ejbActivate`, `ejbPassivate`, and `setSessionContext` methods. The `CartBean` class doesn't use these methods, but it must implement them because they are declared in the `SessionBean` interface. Consequently, these methods are empty in the `CartBean` class. These required methods are hidden in the code fold in the Source Editor. Click the + sign to the left of the code fold to inspect these methods.

You now need to add some private fields to the class declaration. Add the following to the class, after the `private javax.ejb.SessionContext context;` statement:

```
private String customerName;
private String customerId;
private Vector contents;
```

When you add the `private Vector contents` field, the IDE will indicate an error because you have not yet imported the `java.util.vector` library. To add the necessary import statements, place the insertion point anywhere in the class and press Alt-Shift-F to generate the following import statements:

```
import java.util.Vector;
import javax.ejb.CreateException;
```

You can now start adding the create methods and business methods to the class, but for this example you should first create the two helper classes used by the application.

Helper Classes

The `CartBean` session bean has two helper classes: `BookException` and `IdVerifier`. The `BookException` is thrown by the `removeBook` method, and the `IdVerifier` validates the `customerId` in one of the `ejbCreate` methods. Helper classes must reside in the EJB JAR file that contains the enterprise bean class. You will use the IDE's Java Exception wizard to create the `BookException` class, then you will create a new Java class and add the code for the `IdVerifier` helper class

1. Right-click the Source Packages node and choose New→Java Exception.
2. Type `BookException` as the Class Name and `exception` as the Package name, ensuring that Source Packages is selected as the Location, and click Finish.

The IDE generates the `BookException` class and opens the class in the Source Editor. The class has the following code:

```
package exception;

public class BookException extends java.lang.Exception{

     /** Creates a new instance of BookException */
   public BookException() {
   }

   /**
    * Constructs an instance of <code>BookException</code> with
the specified detail message.
    * @param msg the detail message.
    */
   public BookException(String msg) {
      super(msg);
   }
}
```

Now you create the second helper class called `IdVerifier`.

1. Right-click the Source Packages node and choose New→Java Class.
2. Type `IdVerifier` as the Class Name and `util` as the Package name, ensuring that Source Packages is selected as the Location, and click Finish.

3. In the Source Editor, add the following method to the method in the `IdVerifier` class:

```
public boolean validate(String id) {
   boolean result = true;

   for(int i = 0; i < id.length(); i++) {
      if(Character.isDigit(id.charAt(i)) == false)
         result = false;
   }
   return result;
}
```

Now that you have created the helper classes, you can add the create methods and business methods to the `CartBean` class. When you add the methods, the IDE adds the appropriate code to the interfaces.

The ejbCreate Methods

You will now add two create methods to the `CartBean` class. To add the create methods in the IDE, use the Add Create Method contextual menu to generate the methods and add the appropriate code to the interfaces.

1. In the Source Editor, right-click in the body of the `CartBean` class and select EJB Methods →Add Create Method from the contextual menu.
2. Type `create` in the Name field, ensure that the Remote box is selected so that the method is called in the remote interfaces and click Add in the parameter tab.
3. For the new parameter, select `java.lang.String` for the Type, type `person` in the Name field, and then click OK to close each dialog box.
4. The IDE adds the `ejbCreate` method to the `CartBean` class.
5. Now add the following code to the create method:

```
     if (person == null) {
           throw new CreateException("Null person not
allowed.");
     } else {
        customerName = person;
     }
     customerId = "0";
     contents = new Vector();
```

6. You will now add a second create method to the `CartBean` class. Follow steps 1-5 above for generating a create method, this time adding the following two parameters, and in this order:
   - `java.lang.String person`
   - `java.lang.String id`
7. Add the following code to the method you created in step 6:

```
      if (person == null) {
            throw new CreateException("Null person not
allowed.");
      } else {
         customerName = person;
      }
      customerId = "0";
      contents = new Vector();

         IdVerifier idChecker = new IdVerifier();
         if (idChecker.validate(id)) {
         customerId = id;
         } else {
         throw new CreateException("Invalid id: "+ id);
         }
         contents = new Vector();
```

Because an enterprise bean runs inside an EJB container, a client cannot directly instantiate the bean. Only the EJB container can instantiate an enterprise bean. During instantiation, the example program performs the following steps.

1. The client invokes a create method on the home object:

   `Cart shoppingCart = home.create("Duke DeEarl","123");`
2. The EJB container instantiates the enterprise bean.
3. The EJB container invokes the appropriate `ejbCreate` method in `CartBean`.

```
public     void     ejbCreate(java.lang.String     person,
java.lang.String id)throws CreateException {
```

Typically, an `ejbCreate` method initializes the state of the enterprise bean. The preceding `ejbCreate` method, for example, initializes the `customerName` and `customerId` variables by using the arguments passed by the `ejbCreate` method.

An enterprise bean must have one or more `ejbCreate` methods. The signatures of the methods must meet the following requirements:

- The access control modifier must be `public`.
- The return type must be `void`.
- If the bean allows remote access, the arguments must be legal types for the Java Remote Method Invocation (Java RMI) API.
- The modifier cannot be `static` or `final`.

The `throws` clause can include the `javax.ejb.CreateException` and other exceptions that are specific to your application. The `ejbCreate` method usually throws a `CreateException` if an input parameter is invalid.

Business Methods

The primary purpose of a session bean is to run business tasks for the client. The client invokes business methods on the remote object reference that is returned by the `ejbCreate` method. From the client's perspective, the business methods appear to run locally, but they actually run remotely in the session bean. The business methods that a client can invoke are declared in the business interface. The following code snippet shows how the `CartClient` program invokes the business methods:

```
Cart shoppingCart = home.create("Duke DeEarl", "123");
...
shoppingCart.addBook("The Martian Chronicles");
shoppingCart.removeBook("Alice In Wonderland");
bookList = shoppingCart.getContents();
```

The signature of a business method must conform to these rules:

- The method name must not conflict with one defined by the EJB architecture. For example, you cannot call a business method `ejbCreate` or `ejbActivate`.
- The access control modifier must be `public`.
- If the bean allows remote access, the arguments and return types must be legal types for the Java RMI API.
- The modifier must not be `static` or `final`.

When you add business methods in the IDE, you can use the Add Business Method contextual menu to generate the methods. When you do this, the IDE adds the appropriate code to the interfaces. In this example, the business methods are added to the `CartBean` class and the `CartRemoteBusiness` interface.

1. In the Source Editor, right-click in the body of the `CartBean` class and select EJB Methods →Add Business Method from the contextual menu to open the Add Business Method dialog box.
2. Enter `addBook` in the Name field, select `void` as the Return type, and ensure that the Remote box is selected so that the method is called in the remote interfaces. Add a parameter and select `java.lang.String` for the Type, enter `title` in the Name field, and click OK in each dialog box to generate the method.
3. In the Source Editor, edit the `addBook` business method in the `CartBean` class so that the method looks like this:

```
public void addBook(java.lang.String title) {
   contents.add(title);
}
```

Now follow the steps above to create the `removeBook` and `getContents` business methods with the following code:

```
      public void removeBook(java.lang.String title) throws
BookException {
          boolean result = contents.remove(title);
         if (result == false) {
            throw new BookException(title + "not in cart.");
         }
      }

      public Vector getContents() {
         return contents;
      }
```

The `throws` clause can include exceptions that you define for your application. The `removeBook` method, for example, throws the `BookException` if the book is not in the cart. To add the exception in the IDE using the Add Business Method dialog box, click Add in the Exceptions tab and type `BookException`.

When creating the `getContents` business method in the IDE, you can type `Vector` in the Return field in the Add Business Method dialog box.

To indicate a system-level problem, such as the inability to connect to a database, a business method should throw the `javax.ejb.EJBException`. When a business method throws an `EJBException`, the container wraps it in a `RemoteEx-`

`ception`, which is caught by the client. The container will not wrap application exceptions such as `BookException`. Because `EJBException` is a subclass of `RuntimeException`, you do not need to include it in the `throws` clause of the business method.

Managing Your Import Statements

After you have added your create methods and business methods, you need to fix your import statements. Import statements can be added manually, or the IDE can check and fix any import statements in the class. Place the insertion point anywhere in the body of the class in the Source Editor and press Alt-Shift-F to generate the necessary import statements. The IDE removes any unused import statements and adds any missing important statements.

Your import statements for the `CartBean` class should contain the following:

```
import exception.BookException;
import java.util.Vector;
import javax.ejb.CreateException;
import util.IdVerifier;
```

Notice that the IDE adds the import statements for our two helper classes.

You may need to fix or add import statements in the business interfaces. In the CartBean example, you need to fix the imports for the remote business interface (`CartRemoteBusiness`).

Session Bean Interfaces

When you create a session bean in the IDE, the IDE generates the bean structure according to the best practice EJB design patterns. This includes the creation of the bean interfaces. Because the CartBean example uses a remote interface and does not need a local interface, the IDE creates the following interfaces:

- Home interface (`CartRemoteHome`)
- Business interface (`CartRemoteBusiness`)
- Remote interface (`CartRemote`)

A session bean may have a local interface instead of, or in addition to, a remote interface. Generally, a local interface is used when the bean is to be used in the same JVM and a remote interface is used when the bean is to be used in a distributed environment.

Home Interface

A home interface extends the `javax.ejb.EJBHome` interface. For a session bean, the purpose of the home interface is to define the `create` methods that a remote client can invoke. The `CartClient` program, for example, invokes this `create` method:

```
Cart shoppingCart = home.create("Duke DeEarl", "123");
```

Every `create` method in the home interface corresponds to an `ejbCreate` method in the bean class. When you add `create` methods to your session bean using Add Create Method, the corresponding methods are automatically added to the home interface. The signatures of the `ejbCreate` methods in the `CartBean` class follow:

```
public void ejbCreate()
...
public void ejbCreate(java.lang.String person) throws
CreateException
...
public void ejbCreate(java.lang.String person, java.lang.String
id)
   throws CreateException
```

Compare the `ejbCreate` signatures with those of the `create` methods in the `CartRemoteHome` home interface:

```
public interface CartRemoteHome extends javax.ejb.EJBHome {
  cart.CartRemote create() throws
               java.rmi.RemoteException,
javax.ejb.CreateException;
  cart.CartRemote create(java.lang.String person) throws
               java.rmi.RemoteException,
javax.ejb.CreateException;
  cart.CartRemote create(java.lang.String person, String id)
throws
               java.rmi.RemoteException,
javax.ejb.CreateException;
}
```

The signatures of the `ejbCreate` and `create` methods are similar, but they differ in important ways. Defining the signatures of the `create` methods of a home interface follow certain rules.

- The number and types of arguments in a `create` method must match those of its corresponding `ejbCreate` method.
- The arguments and return type of the `create` method must be valid RMI types.
- A `create` method returns the remote interface type of the enterprise bean. (But an `ejbCreate` method returns `void`.)
- The `throws` clause of the `create` method must include the `java.rmi.RemoteException` and the `javax.ejb.CreateException`.

Remote Interface

The remote interface is used when the bean is to be used in a distributed environment. The remote interface extends `javax.ejb.EJBObject` and identifies the business interface whose methods may be invoked from a non-local virtual machine. The remote interface extends the remote business interface, and the bean class only implements the business interface.

Here is the source code for the `CartRemote` remote interface:

```
public interface CartRemote extends javax.ejb.EJBObject,
cart.CartRemoteBusiness{

}
```

The remote interface is empty because you do not need to define your methods in the remote interface. The business methods are defined in the business interface.

Business Interface

The business interface defines the business methods that a remote client can invoke. Here is the source code for the `CartRemoteBusiness` business interface:

```
import java.util.*;
import java.exception.BookException;

public interface CartRemoteBusiness {
  void addBook(java.lang.String title) throws
java.rmi.RemoteException;
  void removeBook(java.lang.String title) throws
                 BookException, java.rmi.RemoteException;
  Vector getContents() throws java.rmi.RemoteException;
}
```

The method definitions in a business interface must follow these rules:

- Each method in the business interface must match a method implemented in the enterprise bean class.
- The signatures of the methods in the business interface must be identical to the signatures of the corresponding methods in the enterprise bean class.
- The arguments and return values must be valid RMI types.

The `throws` clause must include the `java.rmi.RemoteException`.

In this example, the methods in the business interface require you to import libraries. Press Alt-Shift-F to generate the necessary import statments.

Building and Deploying the Application

Now that you have finished creating the Cart EJB module, the next step is to build and deploy the application. You then run the client application to start the session bean. The source files for the example are available in the <*INSTALL*>`/j2eetutorial14/examples/ejb/cart` directory.

1. In the Projects window, right-click the Cart node and select Build Project from the contextual menu.
2. Look at the Output window to ensure the application was built successfully.
3. In the Projects window, right-click the Cart node and select Deploy Project from the contextual menu.

The deployed application is visible in the Runtime window of the IDE. To see the deployed application, expand the EJB Modules node in the Applications node of the server instance. You can undeploy and disable the application in the Runtime window.

The CartClient Application

Now that you have created the session bean, you are ready to run the client application. You can choose to either create the CartClient application or Opening the CartClient Project (page 153) located in the <*INSTALL*>`/j2eetutorial14/examples/ejb/cart/` directory, in which case you need to resolve the references to libraries on the project's classpath.

Creating the CartClient Application

You can create the J2EE application named `CartClient` in the IDE.

1. Choose File→New Project (Ctrl-Shift-N) from the main menu.
2. Choose General from the Categories pane and Java Application in the Projects pane and click Next.
3. Enter `CartClient` as the Project Name, specify a Location for the project and click Finish to create the project.
4. Right-click the CartClient node and choose Properties from the contextual menu.
5. In the Properties dialog box, choose Libraries in the Categories pane and click Add JAR/Folder and add the `j2ee.jar` and `appserv-rt.jar` to the project classpath. Now click Add Project and add the Cart EJB module to the project classpath. Click OK to close the Properties dialog box.
6. Add the following code to the main method:

```
try{
Context ctx = new InitialContext();
Object objRef = ctx.lookup("ejb/CartBean");
CartRemoteHome home =
  (CartRemoteHome)PortableRemoteObject.narrow(objRef,
CartRemoteHome.class);

CartRemote shoppingCart = home.create("Duke DeEarl", "123");

shoppingCart.addBook("The Martian Chronicles");
shoppingCart.addBook("2001 A Space Odyssey");
shoppingCart.addBook("The Left Hand of Darkness");

Vector bookList = new Vector();

bookList = shoppingCart.getContents();

Enumeration enumer = bookList.elements();

while (enumer.hasMoreElements()) {
  String title = (String) enumer.nextElement();

  System.out.println(title);
}

shoppingCart.removeBook("Alice in Wonderland");
shoppingCart.remove();

System.exit(0);

}catch(BookException ex){
  System.err.println("Caught a BookException " +
ex.getMessage());
  System.exit(0);
}catch(Exception ex){
  System.err.println("Caught an unexpected exception: " +
ex.getMessage());
  System.exit(1);
}
```

7. Press Alt-Shift-F to generate the following import statements:

```
import cart.CartRemote;
import cart.CartRemoteHome;
import exception.BookException;
import java.util.Enumeration;
import java.util.Vector;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
```

Opening the CartClient Project

If you open the source code of the CartClient application, you are prompted to resolve library references and add the Cart EJB module and the `j2ee.jar` and `appserv-rt.jar` files to the project classpath.

1. Right-click the CartClient node in the Projects window and choose Resolve Reference Problems. Select the “`Cart” project could not be found` message and click Resolve. In the file chooser, select either the completed Cart project in *<INSTALL>*`/j2eetutorial14/examples/ejb/cart/` or the project you created and click OK.
2. Select the `“appserv-rt.jar” file/folder could not be found` message and click Resolve. Navigate to the `lib` directory in your application server installation, select `appserv-rt.jar`, and click OK. The IDE automatically resolves the location of `j2ee.jar`. Click Close.

Running the CartClient Application

Right-click the CartClient node and select Run Project from the contextual menu. Lines similar to the following are displayed in the Output window of the IDE:

```
run:
... com.sun.corba.ee.spi.logging.LogWrapperBase doLog
INFO: "IOP00710299: (INTERNAL) Successfully created IIOP
listener on the specified host/port: all interfaces/<port>"
The Martian Chronicles
2001 A Space Odyssey
The Left Hand of Darkness
Caught a BookException Alice in Wonderland not in cart.
BUILD SUCCESSFUL (total time: 3 seconds)
```

The HelloService Web Service Example

This example demonstrates a simple web service that generates a response based on information received from the client. `HelloServiceBean` is a stateless session bean that implements a single method, `sayHello`. This method matches the `sayHello` method invoked by the clients. In this section, you will register the HelloService web service with the server and then test the `HelloServiceBean` by running the HelloWebClient JAX-RPC client.

The source code for the HelloService example is located in the *<INSTALL>*`/j2eetutorial/examples/ejb/helloservice/` directory.

Opening the HelloService Example

The HelloService project contains the `HelloServiceBean` class and the *service endpoint interface* (SEI). The `HelloServiceBean` class is located in the `hello` package in the Source Packages node and contains the business method. The `HelloServiceSEI` interface is also located in the `hello` package.

1. Choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to <*INSTALL*>`/j2eetutorial14/examples/ejb/helloservice/`, select the `HelloService` directory, and choose Open Project.
2. Expand the `hello` package under the Source Packages node and open the `HelloServiceSEI` interface in the Source Editor.

Web Service Endpoint Interface

`HelloServiceSEI` is the bean's web service endpoint interface. It provides the client's view of the web service, hiding the stateless session bean from the client. A web service endpoint interface must conform to the rules of a JAX-RPC service definition interface. For a summary of these rules, see Generating and Coding the Service Endpoint Interface and Implementation Class (page 36). Here is the source code for the `HelloService` interface:

```
package hello;

public interface HelloServiceSEI extends java.rmi.Remote {

   public String sayHello(java.lang.String name) throws
java.rmi.RemoteException;
}
```

Stateless Session Bean Implementation Class

The `HelloServiceBean` class implements the `sayHello` method defined by the `HelloServiceSEI` interface. The interface decouples the implementation class from the type of client access. For example, if you added remote and home interfaces to `HelloServiceBean`, the methods of the `HelloServiceBean` class could also be accessed by remote clients. No changes to the `HelloServiceBean` class would be necessary.

The source code for the `HelloServiceBean` class follows:

```
package hello;

import javax.ejb.*;

public class HelloServiceBean implements javax.ejb.SessionBean
{
  private javax.ejb.SessionContext context;

   public String sayHello(java.lang.String name) {

  return "Hello "+ name + " (from HelloServiceBean)";
}

  public void ejbCreate() {}
  public void ejbRemove() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void setSessionContext(javax.ejb.SessionContext
aContext) {
     context = aContext;
  }
}
```

To run the HelloService example, you need to build and deploy the application. You also need to register the web service with the application server before you can run the HelloWebClient application to test the HelloService example.

1. In the Projects window, right-click the HelloService module node and choose Run Project.
2. Expand the Web Services node and right-click the HelloService web service and choose Add to Registry. In the Enter WSDL Url dialog box, ensure that the address is correct and corresponds to your server configuration and click OK.

After you register the web service, the web service is visible in the Runtime window under the Web Services node of the server instance and the HelloService application is visible under the EJB Modules under the Applications node. After you have deployed the application and registered the web service, you can test the web service by running the HelloWebClient application.

Running the HelloWebClient Application

When you run the HelloWebClient application, the HelloWebClient application is deployed to your server and the HelloWebServlet opens in your web browser. This HelloWebClient example already contains the necessary reference to the HelloService web service so you do not need to add it. For this example, it is assumed that your localhost server is running on port 8080. If your server is running on a different port, you will need to edit the following line in the `HelloService.wsdl` file to match your configuration. The `HelloService.wsdl` file is located in the directory `<INSTALL>/j2eetutorial14/examples/ejb/helloservice/HelloWebClient/web/WEB-INF/wsdl/`.

```
    <soap:address
location="http://localhost:8080/webservice/HelloService"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"/>
```

1. Choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to <*INSTALL*>`/j2eetutorial14/examples/ejb/helloservice/`, select the `HelloWebClient` directory, and choose Open Project.
2. In the Projects window, right-click the HelloWebClient project node and choose Run Project. The IDE builds the project, registers the server resources and opens the client in your web browser.
3. When the servlet page opens in your web browser, enter your name in the input box and click Submit to test the web service.

The web page displays the text you input followed by “`(from HelloServiceBean)`”

Other Enterprise Bean Features

The topics that follow apply to session beans and entity beans.

Accessing Environment Entries

Stored in an enterprise bean’s deployment descriptor, an *environment entry* is a name-value pair that allows you to customize the bean’s business logic without changing its source code. An enterprise bean that calculates discounts, for example, might have an environment entry named `Discount Percent`. Before deploying the bean’s application, you could run a development tool to assign `Discount`

`Percent` a value of 0.05 in the bean's deployment descriptor. When you run the application, the bean fetches the 0.05 value from its environment.

In the following code example, the `applyDiscount` method uses environment entries to calculate a discount based on the purchase amount. First, the method locates the environment naming context by invoking `lookup` using the `java:comp/env` parameter. Then it calls `lookup` on the environment to get the values for the `Discount Level` and `Discount Percent` names. For example, if you assign a value of 0.05 to the `Discount Percent` entry, the code will assign 0.05 to the `discountPercent` variable. The `applyDiscount` method, which follows, is in the `CheckerBean` class. The source code for this example is in <*INSTALL*>`/j2eetutorial14/examples/ejb/checker`.

```
public double applyDiscount(double amount) {

  try {

     double discount;

     Context initial = new InitialContext();
        Context environment =
           (Context)initial.lookup("java:comp/env");

     Double discountLevel =
        (Double)environment.lookup("Discount Level");
           Double discountPercent =
              (Double)environment.lookup("Discount Percent");

     if (amount >= discountLevel.doubleValue()) {
        discount = discountPercent.doubleValue() / 100;
     }
     else {
        discount = 0.00;
     }

     return amount * (1.00 - discount);

  } catch (NamingException ex) {
     throw new EJBException("NamingException: "+
        ex.getMessage());
  }
}
```

Comparing Enterprise Beans

A client can determine whether two stateful session beans are identical by invoking the `isIdentical` method:

```
bookCart = home.create("Bill Shakespeare");
videoCart = home.create("Lefty Lee");
...
if (bookCart.isIdentical(bookCart)) {
   // true ... }
if (bookCart.isIdentical(videoCart)) {
   // false ... }
```

Because stateless session beans have the same object identity, the `isIdentical` method always returns `true` when used to compare them.

To determine whether two entity beans are identical, the client can invoke the `isIdentical` method, or it can fetch and compare the beans's primary keys:

```
String key1 = (String)accta.getPrimaryKey();
String key2 = (String)acctb.getPrimaryKey();

if (key1.compareTo(key2) == 0)
   System.out.println("equal");
```

Passing an Enterprise Bean's Object Reference

Suppose that your enterprise bean needs to pass a reference to itself to another bean. You might want to pass the reference, for example, so that the second bean can call the first bean's methods. You can't pass the `this` reference because it points to the bean's instance, which is running in the EJB container. Only the container can directly invoke methods on the bean's instance. Clients access the instance indirectly by invoking methods on the object whose type is the bean's remote interface. It is the reference to this object (the bean's remote reference) that the first bean would pass to the second bean.

A session bean obtains its remote reference by calling the `getEJBObject` method of the `SessionContext` interface. An entity bean would call the `getEJBObject` method of the `EntityContext` interface. These interfaces provide beans with access to the instance contexts maintained by the EJB container. Typically, the bean saves the context in the `setSessionContext` method. The following code fragment shows how a session bean might use these methods.

```
public class WagonBean implements SessionBean {

   SessionContext context;
   ...
   public void setSessionContext(SessionContext aContext) {
  context = aContext;
   }
   ...
   public void passItOn(Basket basket) {
  ...
      basket.copyItems(context.getEJBObject());
   }
```

Using the Timer Service

Applications that model business work flows often rely on timed notifications. The timer service of the enterprise bean container enables you to schedule timed notifications for all types of enterprise beans except for stateful session beans. You can schedule a timed notification to occur at a specific time, after a duration of time, or at timed intervals. For example, you could set timers to go off at 10:30 AM on May 23, in 30 days, or every 12 hours.

When a timer expires (goes off), the container calls the `ejbTimeout` method of the bean's implementation class. The `ejbTimeout` method contains the business logic that handles the timed event. Because `ejbTimeout` is defined by the `javax.ejb.TimedObject` interface, the bean class must implement `TimedObject`.

There are four interfaces in the `javax.ejb` package that are related to timers:

- `TimedObject`
- `Timer`
- `TimerHandle`
- `TimerService`

Creating Timers

To create a timer, the bean invokes one of the `createTimer` methods of the `TimerService` interface. (For details on the method signatures, see the `TimerSer-`

vice API documentation.) When the bean invokes createTimer, the timer service begins to count down the timer duration.

The bean described in the The TimerSessionBean Example (page 162) creates a timer as follows:

```
TimerService timerService = context.getTimerService();
Timer timer = timerService.createTimer(intervalDuration,
  "created timer");
```

In the TimerSessionBean example, createTimer is invoked in a business method, which is called by a client. An entity bean can also create a timer in a business method. If you want to create a timer for each instance of an entity bean, you can code the createTimer call in the bean's ejbCreate method.

Timers are persistent. If the server is shut down (or even crashes), timers are saved and will become active again when the server is restarted. If a timer expires while the server is down, the container will call ejbTimeout when the server is restarted.

A timer for an entity bean is associated with the bean's identity—that is, with a particular instance of the bean. If an entity bean sets a timer in ejbCreate, for example, each bean instance will have its own timer. In contrast, stateless session and message-driven beans do not have unique timers for each instance.

The Date and long parameters of the createTimer methods represent time with the resolution of milliseconds. However, because the timer service is not intended for real-time applications, a callback to ejbTimeout might not occur with millisecond precision. The timer service is for business applications, which typically measure time in hours, days, or longer durations.

Canceling and Saving Timers

Timers can be canceled by the following events:

- When a single-event timer expires, the EJB container calls ejbTimeout and then cancels the timer.
- When an entity bean instance is removed, the container cancels the timers associated with the instance.
- When the bean invokes the cancel method of the Timer interface, the container cancels the timer.

If a method is invoked on a canceled timer, the container throws the `javax.ejb.NoSuchObjectLocalException`.

To save a `Timer` object for future reference, invoke its `getHandle` method and store the `TimerHandle` object in a database. (A `TimerHandle` object is serializable.) To reinstantiate the `Timer` object, retrieve the handle from the database and invoke `getTimer` on the handle. A `TimerHandle` object cannot be passed as an argument of a method defined in a remote or web service interface. In other words, remote clients and web service clients cannot access a bean's `TimerHandle` object. Local clients, however, do not have this restriction.

Getting Timer Information

In addition to defining the `cancel` and `getHandle` methods, the `Timer` interface defines methods for obtaining information about timers:

```
public long getTimeRemaining();
public java.util.Date getNextTimeout();
public java.io.Serializable getInfo();
```

The `getInfo` method returns the object that was the last parameter of the `createTimer` invocation. For example, in the `createTimer` code snippet of the preceding section, this information parameter is a `String` object with the value `created timer`.

To retrieve all of a bean's active timers, call the `getTimers` method of the `TimerService` interface. The `getTimers` method returns a collection of `Timer` objects.

Transactions and Timers

An enterprise bean usually creates a timer within a transaction. If this transaction is rolled back, the timer creation is also rolled back. Similarly, if a bean cancels a timer within a transaction that gets rolled back, the timer cancellation is rolled back. In this case, the timer's duration is reset as if the cancellation had never occurred.

In beans that use container-managed transactions, the `ejbTimeout` method usually has the `RequiresNew` transaction attribute to preserve transaction integrity. With this attribute, the EJB container begins the new transaction before calling ejbTimeout. If the transaction is rolled back, the container will try to call ejb-Timeout at least one more time.

The TimerSessionBean Example

The source code for this example is in the *<INSTALL>*/j2eetutorial14/examples/ejb/timersession/ directory.

TimerSession is a stateless session bean that shows how to set a timer. The implementation class for TimerSessionBean is called TimerSessionBean. In the source code listing of TimerSessionBean that follows, note the myCreateTimer and ejbTimeout methods. Because it's a business method, myCreateTimer is defined in the bean's business interface (TimerSessionRemoteBusiness). The remote interface (TimerSessionRemote) defines the interfaces that can be called by the remote client. In this example, the client invokes myCreateTimer with an interval duration of 30,000 milliseconds. The myCreateTimer method fetches a TimerService object from the bean's SessionContext. Then it creates a new timer by invoking the createTimer method of TimerService. Now that the timer is set, the EJB container will invoke the ejbTimer method of TimerSessionBean when the timer expires—in about 30 seconds. Here's the source code for the TimerSessionBean class:

```
package timer;

import javax.ejb.*;

public class TimerSessionBean implements SessionBean,
TimerSessionRemoteBusiness, TimedObject {
    private SessionContext context;

    public void myCreateTimer(long intervalDuration) {

        System.out.println
            ("TimerSessionBean: start createTimer ");
        TimerService timerService =
            context.getTimerService();
        Timer timer =
            timerService.createTimer(intervalDuration,
             "created timer");
}

    public void ejbTimeout(Timer timer) {
        System.out.println("TimerSessionBean: ejbTimeout ");
    }
    public void setSessionContext(SessionContext aContext) {
        context = aContext;
     }

     public void ejbCreate() {
        System.out.println("TimerSessionBean: ejbCreate");
     }

     public TimerSessionBean() {}
     public void ejbRemove() {}
     public void ejbActivate() {}
     public void ejbPassivate() {}

}
```

Running the TimerSessionBean Example

To run the TimerSessionBean example, you first need to open the TimerClient and the TimerEJB projects in the IDE and build the projects.

1. Choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to `<`*INSTALL*`>/j2eetutorial14/examples/ejb/timersession/`, select the `TimerEJB` directory, and choose Open Project.
2. Choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to `<`*INSTALL*`>/j2eetutorial14/examples/ejb/timersession/`, select the `TimerClient` directory, and choose Open Project. Right-click the TimerClient node in the Projects window and choose Resolve Resource References and locate the `j2ee.jar` and `appserv-rt.jar` files to add to the project classpath. The JAR files can be found in the lib folder of the local installation of the SJS Application Server. You will also need to add the TimerEJB module to the project classpath.
3. Right-click the TimerEJB node in the Projects window and choose Run Project. The TimerEJB application is deployed to the server.
4. Right-click the TimerClient node in the Projects window and choose Run Project.

Lines similar to the following will appear in the Output window:

```
4.5.2005  15:35:43  com.sun.corba.ee.spi.logging.LogWrapper-
Base doLog

INFO:  "IOP00710299:  (INTERNAL)  Successfully  created  IIOP
listener on the specified host/port: all interfaces/2154"

Creating a timer with an interval duration of 30000 ms.
```

```
    BUILD SUCCESSFUL (total time: 2 seconds)
```

The output from the timer appears in the server log file for the localhost in the Output window.

After about 30 seconds you will see lines similar to the following:

```
[#|2005-05-04T15:38:06.320+0200|INFO|sun-appserver-
pe8.1_01|javax.enterprise.system.stream.out|_ThreadID=19;|

TimerSessionBean: start createTimer |#]

[#|2005-05-04T15:38:13.445+0200|INFO|sun-appserver-
pe8.1_01|javax.enterprise.system.stream.out|_ThreadID=19;|

TimerSessionBean: ejbTimeout |#]
```

Handling Exceptions

The exceptions thrown by enterprise beans fall into two categories: system and application.

A *system exception* indicates a problem with the services that support an application. Examples of these problems include the following: a database connection cannot be obtained, an SQL insert fails because the database is full, or a `lookup` method cannot find the desired object. If your enterprise bean encounters a system-level problem, it should throw a `javax.ejb.EJBException`. The container will wrap the `EJBException` in a `RemoteException`, which it passes back to the client. Because the `EJBException` is a subclass of the `RuntimeException`, you do not have to specify it in the `throws` clause of the method declaration. If a system exception is thrown, the EJB container might destroy the bean instance. Therefore, a system exception cannot be handled by the bean's client program; it requires intervention by a system administrator.

An *application exception* signals an error in the business logic of an enterprise bean. There are two types of application exceptions: customized and predefined. A customized exception is one that you've coded yourself, such as the `InsufficentBalanceException` thrown by the `debit` business method of the `SavingsAccountBean` example. The `javax.ejb` package includes several predefined exceptions that are designed to handle common problems. For example, an `ejbCreate` method should throw a `CreateException` to indicate an invalid input parameter. When an enterprise bean throws an application exception, the container does not wrap it in another exception. The client should be able to handle any appli

7

Bean-Managed Persistence Examples

DATA is at the heart of most business applications. In J2EE applications, entity beans represent the business objects that are stored in a database. For entity beans with bean-managed persistence, you must write the code for the database access calls. Although writing this code is an additional responsibility, you will have more control over how the entity bean accesses a database.

This chapter discusses the coding techniques for entity beans with bean-managed persistence. For conceptual information on entity beans, please see What Is an Entity Bean? (page 111).

The SavingsAccountBean Example

The entity bean illustrated in this section represents a simple bank account. The state of the `SavingsAccountBean` enterprise bean is stored in the `savingsac-`

count table of a relational database. The savingsaccount table is created by the following SQL statement:

```
CREATE TABLE savingsaccount
  (id VARCHAR(3)
  CONSTRAINT pk_savingsaccount PRIMARY KEY,
  firstname VARCHAR(24),
  lastname  VARCHAR(24),
  balance   NUMERIC(10,2));
```

The SavingsAccountBean example requires the following code:

- Entity bean class (SavingsAccountBean)
- Home interface (SavingsAccountLocalHome)
- Remote interface (SavingsAccountLocal)

In addition to these standard files, the IDE also creates a business interface (PlayerLocalBusiness) in which it registers business methods. This example also uses the following classes:

- A utility class named InsufficientBalanceException
- A client class called SavingsAccountClient

The source code for this example is in this directory:

```
<INSTALL>/j2eetutorial14/ejb/savingsaccount/
```

Creating the SavingsAccount Project

In the IDE, we have to create a project for the EJB module. We will create a stand-alone EJB module project.

Creating the SavingsAccount Project

1. Choose File→New Project (Ctrl-Shift-N).
2. From the Enterprise template category, select EJB Module and click Next.
3. Type SavingsAccount as the Project Name, specify a location for the project, and click Finish.

Creating the SavingsAccount Enterprise Bean

1. In the Projects window, right-click the SavingsAccount project node and choose New→Entity Bean.
2. In the EJB Name field, type `SavingsAccount`. In the Package field, type `bank`. Set the bean's persistence to Bean and set the bean to only contain remote interfaces. Then click Finish.

Entity Bean Class

The sample entity bean class, `SavingsAccountBean`, is opened in the Source Editor when you create the entity bean. Most of the EJB infrastructure mehtods are hidden in a code fold. Click the + sign at the left of the code fold to inspect these methods.

As you look through the bean class, note that it meets the requirements of any entity bean that uses bean-managed persistence. First, it implements the following:

- `EntityBean` interface
- Zero or more `ejbCreate` and `ejbPostCreate` methods
- Finder methods
- Business methods
- Home methods

In addition, an entity bean class with bean-managed persistence has these requirements:

- The class is defined as `public`.
- The class cannot be defined as `abstract` or `final`.
- It contains an empty constructor.
- It does not implement the `finalize` method.

The EntityBean Interface

The `EntityBean` interface extends the `EnterpriseBean` interface, which extends the `Serializable` interface. The `EntityBean` interface declares a number of methods, such as `ejbActivate` and `ejbLoad`, which you must implement in your entity bean class. These methods are discussed in later sections.

The Database Lookup

Before you can access the database, you must connect to it. When you generate database lookup code in the IDE, a data source and connection pool area automatically added to the project. These resources are configred on the server when you deploy the project.

1. In the Source Editor, right-click anywhere in the body of the `SavingsAccountBean` class and choose Enterprise Resources→Use Database.
2. Change the JNDI Name to `pointbase`, select `jdbc:pointbase://localhost:9092/sun-appserv-samples` in the Connection combo box, and click OK.
3. If prompted for a password, type `pbpublic` in the Password field and click OK.

The IDE inserts the following code in the `SavingsAccountBean` class:

```
private DataSource getPointbase() throws NamingException {
  Context c = new InitialContext();
  return (DataSource)
c.lookup("java:comp/env/jdbc/pointbase");
}
```

Now use the `DataSource` object store a connection to the database in a `Connection` object.

1. In the Source Editor, select the `SavingsAccountBean` class and add the following field to your list of field declarations:

   ```
   private Connection con;
   ```

2. Add a method that makes a connection to the database:

   ```
   private void makeConnection() {
      try {
         con = getPointbase().getConnection();
      } catch (Exception ex) {
         throw new EJBException("Unable to connect to database.
   " +
            ex.getMessage());
      }
   }
   ```

3. Add a method that releases the database connection:

   ```
   private void releaseConnection() {
      try {
         con.close();
   ```

```
   } catch (SQLException ex) {
      throw new EJBException("releaseConnection: " + ex.get-
Message());
   }
}
```

4. Press Alt-Shift-F to generate the following import statements:

```
import java.sql.Connection;
import java.sql.SQLException;
```

Database Access Methods

Now that you have a connection to the database, you need to code the methods that implement the calls to the database.

1. Add the following field declarations to `SavingsAccountBean`:

```
private String id;
private String firstName;
private String lastName;
private BigDecimal balance;
```

2. Generate get and set methods for each of the fields. In the Source Editor, right-click anywhere in `SavingsAccountBean` and choose Refactor→Encapulate Fields. In the dialog box, select the checkbox for the four fields and click Next. Then click Do Refactoring to generate the methods.
3. Add each of the get methods to the remote interface so that they are available to the clients. In the Source Editor, right-click each method's name (for example, `getFirstName`) and choose EJB Methods→Add to Remote Interface.
4. Add the database methods to the entity bean class. You can copy the business methods from the `SavingsAccountBean` class in the <*INSTALL*>`/j2eetutorial14/examples/ejb/savingsaccount/SavingsAccount/src/java` directory. The business methods start with the `// Database Methods` comment on line 251 and end with the `selectInRange` method on line 535.
5. Press Alt-Shift-F to generate the following import statements:

```
import java.math.BigDecimal;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.Collection;
```

The ejbCreate Method

When the client invokes a `create` method, the EJB container invokes the corresponding `ejbCreate` method. Typically, an `ejbCreate` method in an entity bean performs the following tasks:

- Inserts the entity state into the database
- Initializes the instance variables
- Returns the primary key

In the IDE, you can generate `ejbCreate` methods into the bean class and the home interface at the same time.

1. In the Source Editor, right-click anywhere in the body of the `SavingsAccountBean` class and choose EJB Methods→Add Create Method.
2. Use the Add button on the Parameters tab of the dialog box to add the following parameters:
   - `String id`
   - `String firstName`
   - `String lastName`
   - `BigDecimal balance`
3. Leave the default information in the Exceptions tab and click OK.

The IDE inserts an empty `ejbCreate` method and an empty `ejbPostCreate` method into the `SavingsAccountBean` class.

The `ejbCreate` method of `SavingsAccountBean` inserts the entity state into the database by invoking the private `insertRow` method, which issues the SQL `INSERT` statement. Change the contents of the `ejbCreate` method to the following:

```
public String ejbCreate(String id, String firstName,
   String lastName, BigDecimal balance)
   throws CreateException {

   if (balance.signum() == -1)  {
      throw new CreateException
         ("A negative initial balance is not allowed.");
   }

   try {
      insertRow(id, firstName, lastName, balance);
   } catch (Exception ex) {
       throw new EJBException("ejbCreate: " +
          ex.getMessage());
   }

   this.id = id;
   this.firstName = firstName;
   this.lastName = lastName;
   this.balance = balance;

   return id;
}
```

Although the `SavingsAccountBean` class has only one `ejbCreate` method, An enterprise bean can contain multiple `ejbCreate` methods. For an example, see the `CartBean.java` source code in this directory:

    <*INSTALL*>/j2eetutorial14/examples/ejb/cart/

When you write an `ejbCreate` method for an entity bean, be sure to follow these rules:

- The access control modifier must be `public`.
- The return type must be the primary key.
- The arguments must be legal types for the Java RMI API.
- The method modifier cannot be `final` or `static`.

The `throws` clause can include the `javax.ejb.CreateException` and exceptions that are specific to your application. An `ejbCreate` method usually throws a `CreateException` if an input parameter is invalid. If an `ejbCreate` method cannot create an entity because another entity with the same primary key already exists, it should throw a `javax.ejb.DuplicateKeyException` (a subclass of `CreateException`). If a client receives a `CreateException` or a `DuplicateKeyException`, it should assume that the entity was not created.

The state of an entity bean can be directly inserted into the database by an application that is unknown to the Sun Java System Application Server Platform Edition 8. For example, an SQL script might insert a row into the `savingsaccount` table. Although the entity bean for this row was not created by an `ejbCreate` method, the bean can be located by a client program.

The ejbPostCreate Method

For each `ejbCreate` method, your entity bean class must contain an `ejbPostCreate` method. The IDE automatically creates an `ejbPostCreate` method whenever you generate a create method.

The EJB container invokes `ejbPostCreate` immediately after it calls `ejbCreate`. Unlike the `ejbCreate` method, the `ejbPostCreate` method can invoke the `getPrimaryKey` and `getEJBObject` methods of the `EntityContext` interface. For more information on the `getEJBObject` method, see the section Passing an Enterprise Bean's Object Reference (page 160). Often, your `ejbPostCreate` methods will be empty. Leave the `ejbPostCreate` method empty in the `SavingsAccountBean` class.

The signature of an `ejbPostCreate` method must meet the following requirements:

- The number and types of arguments must match a corresponding `ejbCreate` method.
- The access control modifier must be `public`.
- The method modifier cannot be `final` or `static`.
- The return type must be `void`.

The `throws` clause can include the `javax.ejb.CreateException` and exceptions that are specific to your application.

The ejbRemove Method

A client deletes an entity bean by invoking the `remove` method. This invocation causes the EJB container to call the `ejbRemove` method, which deletes the entity state from the database. In the `SavingsAccountBean` class, expand the code fold that contains the EJB infrastructure methods and change the `ejbRemove` method to the following:

```
public void ejbRemove() {
  try {
     deleteRow(id);
  } catch (Exception ex) {
     throw new EJBException("ejbRemove: " +ex.getMessage());
  }
}
```

The `ejbRemove` method invokes a private method named `deleteRow`, which issues an SQL `DELETE` statement.

If the `ejbRemove` method encounters a system problem, it should throw the `javax.ejb.EJBException`. If it encounters an application error, it should throw a `javax.ejb.RemoveException`.

An entity bean can also be removed directly by a database deletion. For example, if an SQL script deletes a row that contains an entity bean state, then that entity bean is removed.

The ejbLoad and ejbStore Methods

If the EJB container needs to synchronize the instance variables of an entity bean with the corresponding values stored in a database, it invokes the `ejbLoad` and `ejbStore` methods. The `ejbLoad` method refreshes the instance variables from the database, and the `ejbStore` method writes the variables to the database. The client cannot call `ejbLoad` and `ejbStore`.

If a business method is associated with a transaction, the container invokes `ejb-Load` before the business method executes. Immediately after the business method executes, the container calls `ejbStore`. Because the container invokes `ejbLoad` and `ejbStore`, you do not have to refresh and store the instance variables in your business methods. The `SavingsAccountBean` class relies on the container to synchronize the instance variables with the database. Therefore, the business methods of `SavingsAccountBean` should be associated with transactions.

If the `ejbLoad` and `ejbStore` methods cannot locate an entity in the underlying database, they should throw the `javax.ejb.NoSuchEntityException`. This exception is a subclass of `EJBException`. Because `EJBException` is a subclass of `RuntimeException`, you do not have to include it in the `throws` clause. When `NoSuchEntityException` is thrown, the EJB container wraps it in a `RemoteEx-ception` before returning it to the client.

In the Source Editor, change the `ejbLoad` and `ejbStore` methods to the following:

```
public void ejbLoad() {
  try {
     loadRow();
  } catch (Exception ex) {
     throw new EJBException("ejbLoad: " + ex.getMessage());
  }
}

public void ejbStore() {
  try {
     storeRow();
  } catch (Exception ex) {
     throw new EJBException("ejbStore: " + ex.getMessage());
  }
}
```

In the `SavingsAccountBean` class, `ejbLoad` invokes the `loadRow` method, which issues an SQL `SELECT` statement and assigns the retrieved data to the instance variables. The `ejbStore` method calls the `storeRow` method, which stores the instance variables in the database using an SQL `UPDATE` statement.

The Finder Methods

The finder methods allow clients to locate entity beans. The `SavingsAccountClient` program locates entity beans using three finder methods:

```
SavingsAccountRemote jones = home.findByPrimaryKey("836");
...
Collection c = home.findByLastName("Smith");
...
Collection c = home.findInRange(new BigDecimal("20.00"),
       new BigDecimal("99.00"));
```

For every finder method available to a client, the entity bean class must implement a corresponding method that begins with the prefix `ejbFind`. The `SavingsAccountBean` class, for example, implements two optional finder methods: `ejbFindByLastName` and `ejbFindInRange`.

1. In the Source Editor, right-click anywhere in the body of the `SavingsAccountBean` class and choose EJB Methods→Add Finder Method.
2. In the Name field, type `findByLastName`. Leave the Return Cardinality set to Many and the Remote interface selected. Use the Parameters tab to add a `String lastName` parameter. Then click OK to generate the finder method in both the bean class and the home interface.
3. Edit the ejbFindByLastName method as follows:

```
public Collection ejbFindByLastName(String lastName)
     throws FinderException {
  Collection result;
  try {
     result = selectByLastName(lastName);
  } catch (Exception ex) {
     throw new EJBException("ejbFindByLastName " +
        ex.getMessage());
  }
  return result;
}
```

4. Repeat steps 1-3 to create the following `ejbFindInRange` method:

```
public Collection ejbFindInRange(BigDecimal low,
      BigDecimal high) throws FinderException {
   Collection result;
   try {
      result = selectInRange(low, high);
   } catch (Exception ex) {
      throw new EJBException("ejbFindInRange: "
            + ex.getMessage());
   }
   return result;
}
```

The finder methods that are specific to your application, such as `ejbFindByLastName` and `ejbFindInRange`, are optional, but the `ejbFindByPrimaryKey` method is required. As its name implies, the `ejbFindByPrimaryKey` method accepts as an argument the primary key, which it uses to locate an entity bean. In the `SavingsAccountBean` class, the primary key is the `id` variable. Edit the `ejbFindByPrimaryKey` method as follows:

```
public String ejbFindByPrimaryKey(String aKey)
     throws FinderException {

   boolean result;

   try {
      result = selectByPrimaryKey(aKey);
   } catch (Exception ex) {
      throw new EJBException("ejbFindByPrimaryKey: " +
         ex.getMessage());
   }

   if (result) {
      return aKey;
   }
   else {
      throw new ObjectNotFoundException
         ("Row for id " + aKey + " not found.");
   }
}
```

The `ejbFindByPrimaryKey` method may look strange to you, because it uses a primary key for both the method argument and the return value. However, remember that the client does not call `ejbFindByPrimaryKey` directly. It is the EJB container that calls the `ejbFindByPrimaryKey` method. The client invokes the `findByPrimaryKey` method, which is defined in the home interface.

The following list summarizes the rules for the finder methods that you implement in an entity bean class with bean-managed persistence:

- The `ejbFindByPrimaryKey` method must be implemented.
- A finder method name must start with the prefix `ejbFind`.
- The access control modifier must be `public`.
- The method modifier cannot be `final` or `static`.
- The arguments and return type must be legal types for the Java RMI API. (This requirement applies only to methods defined in a remote—and not a local—home interface.)
- The return type must be the primary key or a collection of primary keys.

The `throws` clause can include the `javax.ejb.FinderException` and exceptions that are specific to your application. If a finder method returns a single primary key and the requested entity does not exist, the method should throw the `javax.ejb.ObjectNotFoundException` (a subclass of `FinderException`). If a finder method returns a collection of primary keys and it does not find any objects, it should return an empty collection.

The Business Methods

The business methods contain the business logic that you want to encapsulate within the entity bean. Usually, the business methods do not access the database, and this allows you to separate the business logic from the database access code.

First you need to create a special Java exception class that your business methods will use.

1. In the Projects window, right-click the SavingsAccount project node and choose New→File/Folder.
2. From the Java Classes category, select the Java Exception template and click Next.
3. Name the class `InsufficientBalanceException`, place it in the `bank` package, and choose Finish. The IDE creates the class and opens it in the

Source Editor. You could customize the way the exception is handled. For our purposes, we will just use the basic code provided by the template.

Now you can add the business methods.

1. In the Source Editor, right-click anywhere in the body of the `SavingsAccountBean` class and choose EJB Methods→Add Business Method.
2. In the Name field, type `debit`. Leave `void` in the Return Type combo box. In the Parameters tab, use the Add button to add a `BigDecimal amount` parameter. In the Exceptions tab, use the Add button to add an `InsufficientBalanceException` exception. Then click OK to generate the method.
3. Edit the body of the the debit method as follows:

```
public void debit(BigDecimal amount)
     throws InsufficientBalanceException {
   if (balance.compareTo(amount) == -1) {
     throw new InsufficientBalanceException();
   }
   balance = balance.subtract(amount);
}
```

4. Repeat steps 1-3 to create the following business method. Make sure you set the return types, exceptions, and parameters correctly:

```
public void credit(BigDecimal amount) {
   balance = balance.add(amount);
}
```

The `SavingsAccountClient` program invokes the business methods as follows:

```
BigDecimal zeroAmount = new BigDecimal("0.00");
SavingsAccount duke = home.create("123", "Duke", "Earl",
    zeroAmount);
...
duke.credit(new BigDecimal("88.50"));
duke.debit(new BigDecimal("20.25"));
BigDecimal balance = duke.getBalance();
```

The requirements for the signature of a business method are the same for session beans and entity beans:

- The method name must not conflict with a method name defined by the EJB architecture. For example, you cannot call a business method `ejbCreate` or `ejbActivate`.
- The access control modifier must be `public`.
- The method modifier cannot be `final` or `static`.

- The arguments and return types must be legal types for the Java RMI API. This requirement applies only to methods defined in a remote—and not a local—home interface.

The `throws` clause can include the exceptions that you define for your application. The `debit` method, for example, throws the `InsufficientBalanceException`. To indicate a system-level problem, a business method should throw the `javax.ejb.EJBException`.

The Home Methods

A home method contains the business logic that applies to all entity beans of a particular class. In contrast, the logic in a business method applies to a single entity bean, an instance with a unique identity. During a home method invocation, the instance has neither a unique identity nor a state that represents a business object. Consequently, a home method must not access the bean's persistence state (instance variables). (For container-managed persistence, a home method also must not access relationships.)

Typically, a home method locates a collection of bean instances and invokes business methods as it iterates through the collection. This approach is taken by the `ejbHomeChargeForLowBalance` method of the `SavingsAccountBean` class. The `ejbHomeChargeForLowBalance` method applies a service charge to all savings accounts that have balances less than a specified amount. The method locates these accounts by invoking the `findInRange` method. As it iterates through the collection of `SavingsAccount` instances, the `ejbHomeChargeForLowBalance` method checks the balance and invokes the `debit` business method.

1. In the Source Editor, right-click anywhere in the body of the `SavingsAccountBean` class and choose EJB Methods→Add Home Method.
2. In the Name field, type `chargeForLowBalance`. Leave `void` in the Return Type combo box. In the Parameters tab, use the Add button to add the following parameters:
   - `BigDecimal minimumBalance`
   - `BigDecimal charge`
3. In the Exceptions tab, use the Add button to add an `InsufficientBalanceException` exception. Then click OK to generate the method.
4. Edit the body of the the method as follows:

```
public void ejbHomeChargeForLowBalance(
     BigDecimal minimumBalance, BigDecimal charge)
```

```
         throws InsufficientBalanceException {

      try {
         SavingsAccountRemoteHome home =
         (SavingsAccountRemoteHome)context.getEJBHome();
         Collection c = home.findInRange(new BigDecimal("0.00"),
         minimumBalance.subtract(new BigDecimal("0.01")));

           Iterator i = c.iterator();

           while (i.hasNext()) {
              SavingsAccountRemote account =
                 (SavingsAccountRemote)i.next();
              if (account.getBalance().compareTo(charge) == 1) {
                 account.debit(charge);
              }
           }

      } catch (Exception ex) {
           throw new EJBException("ejbHomeChargeForLowBalance: "
            + ex.getMessage());
       }
   }
```

The home interface defines a corresponding method named `chargeForLowBalance` (see Home Method Definitions, page 183). Because the interface provides the client view, the `SavingsAccountClient` program invokes the home method as follows:

```
SavingsAccountRemoteHome home;
...
home.chargeForLowBalance(new BigDecimal("10.00"),
  new BigDecimal("1.00"));
```

In the entity bean class, the implementation of a home method must adhere to these rules:

- A home method name must start with the prefix `ejbHome`.
- The access control modifier must be `public`.
- The method modifier cannot be `static`.

The `throws` clause can include exceptions that are specific to your application; it must not throw the `java.rmi.RemoteException`.

Home Interface

The home interface defines the `create`, finder, and home methods. The `SavingsAccountRemoteHome` interface follows:

```
import java.util.Collection;
import java.math.BigDecimal;
import java.rmi.RemoteException;
import javax.ejb.*;

public interface SavingsAccountRemoteHome extends EJBHome {

  SavingsAccountRemote create(String id, String firstName,
     String lastName, BigDecimal balance)
     throws RemoteException, CreateException;

  SavingsAccountRemote findByPrimaryKey(String key)
     throws FinderException, RemoteException;

  Collection findByLastName(String lastName)
     throws FinderException, RemoteException;

  Collection findInRange(BigDecimal low, BigDecimal high)
     throws FinderException, RemoteException;

  void chargeForLowBalance(BigDecimal minimumBalance,
     BigDecimal charge)throws InsufficientBalanceException,
     RemoteException;
}
```

Note: Since you used simple names of classes to generate the home methods and finder methods, you have to open the home interface and press Alt-Shift-F to generate the necessary import statements.

create Method Definitions

Each `create` method in the home interface must conform to the following requirements:

- It must have the same number and types of arguments as its matching `ejbCreate` method in the enterprise bean class.
- It must return the remote interface type of the enterprise bean.
- The `throws` clause must include the exceptions specified by the `throws` clause of the corresponding `ejbCreate` and `ejbPostCreate` methods.
- The `throws` clause must include the `javax.ejb.CreateException.`
- If the method is defined in a remote—and not a local—home interface, then the `throws` clause must include the `java.rmi.RemoteException`.

Finder Method Definitions

Every finder method in the home interface corresponds to a finder method in the entity bean class. The name of a finder method in the home interface begins with `find`, whereas the corresponding name in the entity bean class begins with `ejbFind`. For example, the `SavingsAccountRemoteHome` class defines the `findByLastName` method, and the `SavingsAccountBean` class implements the `ejbFindByLastName` method. The rules for defining the signatures of the finder methods of a home interface follow.

- The number and types of arguments must match those of the corresponding method in the entity bean class.
- The return type must be the entity bean's remote interface type or a collection of those types.
- The exceptions in the `throws` clause must include those of the corresponding method in the entity bean class.
- The `throws` clause must contain the `javax.ejb.FinderException`.
- If the method is defined in a remote—and not a local—home interface, then the `throws` clause must include the `java.rmi.RemoteException`.

Home Method Definitions

Each home method definition in the home interface corresponds to a method in the entity bean class. In the home interface, the method name is arbitrary, provided that it does not begin with `create` or `find`. In the bean class, the matching method name begins with `ejbHome`. For example, in the `SavingsAccountBean` class the name is `ejbHomeChargeForLowBalance`, but in the `SavingsAccountRemoteHome` interface the name is `chargeForLowBalance`.

The home method signature must follow the same rules specified for finder methods in the preceding section (except that a home method does not throw a `FinderException`).

Remote Interface

The remote interface usually extends `javax.ejb.EJBObject` and defines the business methods that a remote client can invoke. Because the IDE enforces best design pratcices, it registers all of an entity bean's business methods in a business interface. The remote interface then extends the remote business interface, and the bean class only implements the business interface.

The remote interface is therefore empty:

```
package bank;
public interface SavingsAccountRemote extends
    javax.ejb.EJBObject,bank.SavingsAccountRemoteBusiness {

}
```

Here is the `SavingsAccountRemoteBusiness` interface:

```
import java.rmi.RemoteException;
import java.math.BigDecimal;

public interface SavingsAccountRemote extends EJBObject {

  void debit(BigDecimal amount)
     throws InsufficientBalanceException, RemoteException;

  void credit(BigDecimal amount)
     throws RemoteException;

  String getId()
     throws RemoteException;

  String getFirstName()
     throws RemoteException;

  String getLastName()
     throws RemoteException;
```

```
    BigDecimal getBalance()
       throws RemoteException;
 }
```

Note: Since you used simple names of classes to generate the business methods, you have to open the business interface and press Alt-Shift-F to generate the necessary import statements.

The requirements for the method definitions in a remote interface are the same for session beans and entity beans:

- Each method in the remote interface must match a method in the enterprise bean class.
- The signatures of the methods in the remote interface must be identical to the signatures of the corresponding methods in the enterprise bean class.
- The arguments and return values must be valid RMI types.
- The `throws` clause must include `java.rmi.RemoteException`.

A local interface has the same requirements, with the following exceptions:

- The arguments and return values are not required to be valid RMI types.
- The `throws` clause does not include `java.rmi.RemoteException`.

Running the SavingsAccount Example

Before you run this example, you have to create the database and deploy the `SavingsAccount.jar` file.

Creating the Sample Database

The instructions that follow explain how to use the `SavingsAccountBean` example with PointBase, the database software that is included in the Application Server bundle.

1. In the IDE, choose Tools→PointBase Database→Start Local PointBase Database.

2. Create the database tables by running the `create.sql` script.
   a. Make sure that the `appsrv.root` property in your `<INSTALL>/j2eetutorial14/examples/` file points to the location of your local Application Server installation.
   b. In a terminal window, go to this directory:

      ```
      <INSTALL>/j2eetutorial14/examples/ejb/savingsaccount/
      ```
   c. Type the following command, which runs the `create.sql` script:

      ```
      asant -buildfile create-db.xml
      ```
3. In the Runtime window, expand the Databases node, right-click the `jdbc:pointbase:server://localhost:9092/sun-appserv-samples` node, and choose Connect. Type `pbpublic` as the password and click OK. Once the connection is established, expand the connection node's Tables node. There should be a node for the `SAVINGSACCOUNT` table.

Deploying the Application

1. In the Projects window, right-click the SavingsAccount project node and choose Deploy Project. The IDE does the following
   - Builds the EJB module
   - Starts the application server if it is not already started
   - Configures the data source and connection pool on the application server
   - Deploys the EJB module to the application server
2. In the Runtime window, expand Servers→Sun Java System Application Server→Applications→EJB Modules and verify that the `SavingsAccount` EJB module exists on the server.

Running the Client

The source code for the `SavingsAccountClient` project is in the <*INSTALL*>`/j2eetutorial14/examples/ejb/savingsaccount` directory. When you open the project, you have to resolve the references to libraries on the project's classpath.

1. Choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to <*INSTALL*>`/j2eetutorial14/examples/ejb/savingsaccount/`, select the `SavingsAccountClient` directory, and choose Open Project.

2. The project needs to know the location of some JAR files on its classpath and the SavingsAccount project. Right-click the SavingsAccountClient project and choose Resolve Reference Problems. Select the `“SavingsAccount” project could not be found message` and click Resolve. In the file chooser, select either the completed SavingsAccount project in <*INSTALL*>`/j2eetutorial14/examples/ejb/savingsaccount/` or the project you created and click OK.
3. Select the `“appserv-rt.jar” file/folder could not be found` message and click Resolve. Navigate to the `lib` directory in your application server installation, select `appserv-rt.jar`, and click OK. The IDE automatically resolves the location of `j2ee.jar`. Click Close.
4. Right-click the SavingsAccountClient project and choose Run Project. The client should display the following lines:

```
balance = 68.25
balance = 32.55
456: 44.77
730: 19.54
268: 100.07
836: 32.55
456: 44.77
4
7
```

To modify this example, see the instructions in Modifying the J2EE Application (page 139).

Mapping Table Relationships for Bean-Managed Persistence

In a relational database, tables can be related by common columns. The relationships between the tables affect the design of their corresponding entity beans. The entity beans discussed in this section are backed up by tables with the following types of relationships:

- One-to-one
- One-to-many
- Many-to-many

One-to-One Relationships

In a one-to-one relationship, each row in a table is related to a single row in another table. For example, in a warehouse application, a `storagebin` table might have a one-to-one relationship with a `widget` table. This application would model a physical warehouse in which each storage bin contains one type of widget and each widget resides in one storage bin.

Figure 7–1 illustrates the `storagebin` and `widget` tables. Because the `storage-binid` uniquely identifies a row in the `storagebin` table, it is that table's primary key. The `widgetid` is the primary key of the `widget` table. The two tables are related because the `widgetid` is also a column in the `storagebin` table. By referring to the primary key of the `widget` table, the `widgetid` in the `storagebin` table identifies which widget resides in a particular storage bin in the warehouse. B .5(7(g2pa)u)5.1(s)((B .5(of t5.4(e)]TJ/F1 1 Tf9 0 0 913.998 481.16 Tm0 Tc0 Tw[(wic checks in the application code. The `storagebin` table has a referential constraint named `fk_widgetid`:

```
CREATE TABLE storagebin
   (storagebinid VARCHAR(3)
    CONSTRAINT pk_storagebin PRIMARY KEY,
    widgetid VARCHAR(3),
    quantity INTEGER,
    CONSTRAINT fk_widgetid
    FOREIGN KEY (widgetid)
     REFERENCES widget(widgetid));
```

The source code for the following example is in this directory:

```
<INSTALL>/j2eetutorial14/examples/ejb/storagebin/
```

To open the project, choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to *<INSTALL>*/j2eetutorial14/examples/ejb/storagebin/, select the `StorageBin` directory, and choose Open Project.

The `StorageBinBean` and `WidgetBean` classes illustrate the one-to-one relationship of the `storagebin` and `widget` tables. The `StorageBinBean` class contains variables for each column in the `storagebin` table, including the foreign key, `widgetId`:

```
private String storageBinId;
private String widgetId;
private int quantity;
```

The `ejbFindByWidgetId` method of the `StorageBinBean` class returns the `storageBinId` that matches a given `widgetId`:

```
public String ejbFindByWidgetId(String widgetId)
   throws FinderException {

   String storageBinId;

   try {
      storageBinId = selectByWidgetId(widgetId);
    } catch (Exception ex) {
        throw new EJBException("ejbFindByWidgetId: " +
           ex.getMessage());
    }

   if (storageBinId == null) {
      throw new ObjectNotFoundException
```

```
            ("Row for widgetId " + widgetId + " not found.");
      }
      else {
         return storageBinId;
      }
   }
```

The `ejbFindByWidgetId` method locates the `widgetId` by querying the database in the `selectByWidgetId` method:

```
   private String selectByWidgetId(String widgetId)
      throws SQLException {

      String storageBinId;

      makeConnection();
      String selectStatement =
            "select storagebinid " +
            "from storagebin where widgetid = ? ";
      PreparedStatement prepStmt =
            con.prepareStatement(selectStatement);
      prepStmt.setString(1, widgetId);

      ResultSet rs = prepStmt.executeQuery();

      if (rs.next()) {
         storageBinId = rs.getString(1);
      }
      else {
         storageBinId = null;
      }

      prepStmt.close();
      releaseConnection();
      return storageBinId;
   }
```

To find out in which storage bin a widget resides, the `StorageBinClient` program calls the `findByWidgetId` method:

```
   String widgetId = "777";
   StorageBin storageBin =
      storageBinHome.findByWidgetId(widgetId);
   String storageBinId = (String)storageBin.getPrimaryKey();
   int quantity = storageBin.getQuantity();
```

Running the StorageBinBean Example

1. In the IDE, choose Tools→PointBase Database→Start Local PointBase Database.
2. Create the database tables by running the `create.sql` script.
   a. Make sure that the `appsrv.root` property in your `<INSTALL>/j2eetutorial14/examples/` file points to the location of your local Application Server installation.
   b. In a terminal window, go to this directory:
   ```
   <INSTALL>/j2eetutorial14/examples/ejb/storagebin/
   ```
   c. Type the following command, which runs the `create.sql` script:
   ```
   asant -buildfile create-db.xml
   ```
3. Choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to `<INSTALL>/j2eetutorial14/examples/ejb/storagebin/`, select the `StorageBinClient` directory, and choose Open Project.
4. The project needs to know the location of some JAR files on its classpath and the StorageBin project. Right-click the StorageBinClient project and choose Resolve Reference Problems. Select the `"StorageBin" project could not be found message` and click Resolve. In the file chooser, select either the completed StorageBin project in `<INSTALL>/j2eetutorial14/examples/ejb/storagebin/` or the project you created and click OK.
5. Select the `"appserv-rt.jar" file/folder could not be found` message and click Resolve. Navigate to the `lib` directory in your application server installation, select `appserv-rt.jar`, and click OK. The IDE automatically resolves the location of `j2ee.jar`. Click Close.
6. Right-click the StorageBin project and choose Deploy Project. The IDE builds the project, deploys the EJB module, and registers a JDBC connection pool and database resource for the project.
7. Right-click the StorageBinClient project and choose Run Project. The client should display the following:
   ```
   ...
   777 388 500 1.0 Duct Tape
   ...
   ```

One-to-Many Relationships

If the primary key in a parent table matches multiple foreign keys in a child table, then the relationship is one-to-many. This relationship is common in database applications. For example, an application for a sports league might access a `team` table and a `player` table. Each team has multiple players, and each player belongs to a single team. Every row in the child table (`player`) has a foreign key identifying the player's team. This foreign key matches the `team` table's primary key.

The sections that follow describe how you might implement one-to-many relationships in entity beans. When designing such entity beans, you must decide whether both tables are represented by entity beans, or only one.

A Helper Class for the Child Table

Not every database table needs to be mapped to an entity bean. If a database table doesn't represent a business entity, or if it stores information that is contained in another entity, then you should use a helper class to represent the table. In an online shopping application, for example, each order submitted by a customer can have multiple line items. The application stores the information in the database tables shown by Figure 7–2.

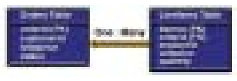


Figure 7–2 One-to-Many Relationship: Order and Line Items

Not only does a line item belong to an order, but it also does not exist without the order. Therefore, the `lineitems` table should be represented with a helper class and not with an entity bean. Using a helper class in this case is not required, but doing so might improve performance because a helper class uses fewer system resources than does an entity bean.

The source code for the following example is in this directory:

```
<INSTALL>/j2eetutorial14/examples/ejb/order/
```

To open the project, choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to `<INSTALL>/j2eetutorial14/examples/ejb/order/`, select the `Order` directory, and choose Open Project.

The `LineItem` and `OrderBean` classes show how to implement a one-to-many relationship using a helper class (`LineItem`). The instance variables in the `LineItem` class correspond to the columns in the `lineitems` table. The `itemNo` variable matches the primary key for the `lineitems` table, and the `orderId` variable represents the table's foreign key. Here is the source code for the `LineItem` class:

```
public class LineItem implements java.io.Serializable {

   String productId;
   int quantity;
   double unitPrice;
   int itemNo;
   String orderId;


   public LineItem(String productId, int quantity,
     double unitPrice, int itemNo, String orderId) {

      this.productId = productId;
      this.quantity = quantity;
      this.unitPrice = unitPrice;
      this.itemNo = itemNo;
      this.orderId = orderId;
   }

   public String getProductId() {
      return productId;
   }

   public int getQuantity() {
      return quantity;
   }

   public double getUnitPrice() {
      return unitPrice;
   }

   public int getItemNo() {
      return itemNo;
   }

   public String getOrderId() {
      return orderId;
   }
}
```

The `OrderBean` class contains an `ArrayList` variable named `lineItems`. Each element in the `lineItems` variable is a `LineItem` object. The `lineItems` variable is passed to the `OrderBean` class in the `ejbCreate` method. For every `LineItem` object in the `lineItems` variable, the `ejbCreate` method inserts a row into the `lineitems` table. It also inserts a single row into the `orders` table. The code for the `ejbCreate` method follows:

```
public String ejbCreate(String orderId, String customerId,
    String status, double totalPrice, ArrayList lineItems)
    throws CreateException {

    try {
       insertOrder(orderId, customerId, status, totalPrice);
       for (int i = 0; i < lineItems.size(); i++) {
          LineItem item = (LineItem)lineItems.get(i);
          insertItem(item);
       }
    } catch (Exception ex) {
        throw new EJBException("ejbCreate: " +
           ex.getMessage());
    }

    this.orderId = orderId;
    this.customerId = customerId;
    this.status = status;
    this.totalPrice = totalPrice;
    this.lineItems = lineItems ;

    return orderId;
}
```

The `OrderClient` program creates and loads an `ArrayList` of `LineItem` objects. The program passes this `ArrayList` to the entity bean when it invokes the `create` method:

```
ArrayList lineItems = new ArrayList();
lineItems.add(new LineItem("p23", 13, 12.00, 1, "123"));
lineItems.add(new LineItem("p67", 47, 89.00, 2, "123"));
lineItems.add(new LineItem("p11", 28, 41.00, 3, "123"));
...
OrderRemote duke = home.create("123", "c44", "open",
   totalItems(lineItems), lineItems);
```

Other methods in the `OrderBean` class also access both database tables. The `ejbRemove` method, for example, not only deletes a row from the `orders` table but also deletes all corresponding rows in the `lineitems` table. The `ejbLoad` and `ejbStore` methods synchronize the state of an `OrderBean` instance, including the `lineItems` `ArrayList`, with the `orders` and `lineitems` tables.

The `ejbFindByProductId` method enables clients to locate all orders that have a particular product. This method queries the `lineitems` table for all rows with a specific `productId`. The method returns a `Collection` of `Order` objects. The `OrderClient` program iterates through the `Collection` and prints the primary key of each order:

```
Collection c = home.findByProductId("p67");
Iterator i=c.iterator();
while (i.hasNext()) {
   OrderRemote order = (OrderRemote)i.next();
   String id = (String)order.getPrimaryKey();
   System.out.println(id);
}
```

Running the OrderBean Example

1. In the IDE, choose Tools→PointBase Database→Start Local PointBase Database.
2. Create the database tables by running the `create.sql` script.
   a. Make sure that the `appsrv.root` property in your `<`*`INSTALL`*`>/j2eetutorial14/examples/` file points to the location of your local Application Server installation.
   b. In a terminal window, go to this directory:

   `<`*`INSTALL`*`>/j2eetutorial14/examples/ejb/order/`

   c. Type the following command, which runs the `create.sql` script:

   ```
   asant -buildfile create-db.xml
   ```

3. Choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to *<INSTALL>*`/j2eetutorial14/examples/ejb/order/`, select the `OrderClient` directory, and choose Open Project.
4. The project needs to know the location of some JAR files on its classpath and the Order project. Right-click the OrderClient project and choose Resolve Reference Problems. Select the `"Order" project could not be found message` and click Resolve. In the file chooser, select either the completed Order project in *<INSTALL>*`/j2eetutorial14/examples/ejb/order/` or the project you created and click OK.
5. Select the `"appserv-rt.jar" file/folder could not be found` message and click Resolve. Navigate to the `lib` directory in your application server installation, select `appserv-rt.jar`, and click OK. The IDE automatically resolves the location of `j2ee.jar`. Click Close.
6. Right-click the Order project and choose Deploy Project. The IDE builds the project, deploys the EJB module, and registers a JDBC connection pool and database resource for the project.
7. Right-click the OrderClient project and choose Run Project. The client should display the following:

   ```
   ...
   123 1 p23 12.0
   123 2 p67 89.0
   123 3 p11 41.0

   123
   456
   ```

An Entity Bean for the Child Table

You should consider building an entity bean for a child table under the following conditions:

- The information in the child table is not dependent on the parent table.
- The business entity of the child table could exist without that of the parent table.
- The child table might be accessed by another application that does not access the parent table.

These conditions exist in the following scenario. Suppose that each sales representative in a company has multiple customers and that each customer has only one sales representative. The company tracks its sales force using a database application. In the database, each row in the `salesrep` table (parent) matches multiple rows in the `customer` table (child). Figure 7–3 illustrates this relationship.

Figure 7–3 One-to-Many Relationship: Sales Representative and Customers

The `SalesRepBean` and `CustomerBean` entity bean classes implement the one-to-many relationship of the `sales` and `customer` tables.

The source code for this example is in this directory:

```
<INSTALL>/j2eetutorial14/examples/ejb/salesrep/
```

To open the project, choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to <*INSTALL*>/j2eetutorial14/examples/ejb/salesrep/, select the SalesRep directory, and choose Open Project.

The `SalesRepBean` class contains a variable named `customerIds`, which is an `ArrayList` of `String` elements. These `String` elements identify which customers belong to the sales representative. Because the `customerIds` variable reflects this relationship, the `SalesRepBean` class must keep the variable up-to-date.

The `SalesRepBean` class instantiates the `customerIds` variable in the `setEntityContext` method and not in `ejbCreate`. The container invokes `setEntityContext` only once—when it creates the bean instance—thereby ensuring that `customerIds` is instantiated only once. Because the same bean instance can assume different identities during its life cycle, instantiating `customerIds` in `ejbCreate` might cause multiple and unnecessary instantiations. Therefore, the

SalesRepBean class instantiates the customerIds variable in setEntityContext:

```
public void setEntityContext(EntityContext context) {

   this.context = context;
   customerIds = new ArrayList();

   try {
      Context initial = new InitialContext();
      Object objref =
         initial.lookup("java:comp/env/ejb/Customer");

      customerHome =

(CustomerRemoteHome)PortableRemoteObject.narrow(objref,
          CustomerRemoteHome.class);
   } catch (Exception ex) {
      throw new EJBException("setEntityContext: " +
         ex.getMessage());
   }
}
```

Invoked by the ejbLoad method, loadCustomerIds is a private method that refreshes the customerIds variable. There are two approaches to coding a method such as loadCustomerIds: fetch the identifiers from the customer database table, or get them from the CustomerBean entity bean. Fetching the identifiers from the database might be faster, but it exposes the code in the SalesRepBean class to the CustomerBean bean's underlying database table. In the future, if you were to change the CustomerBean bean's table (or move the bean to a different Application Server), you might need to change the SalesRepBean code. But if the SalesRepBean class gets the identifiers from the CustomerBean entity bean, no coding changes would be required. The two approaches present a trade-off: performance versus flexibility. The SalesRepBean example opts for flexibility, loading the customerIds variable by calling the findBySalesRep and getPrimaryKey methods of CustomerBean. Here is the code for the loadCustomerIds method:

```
private void loadCustomerIds() {

    customerIds.clear();

    try {
       Collection c = customerHome.findBySalesRep(salesRepId);
       Iterator i=c.iterator();
```

```
         while (i.hasNext()) {
        CustomerRemote customer = (CustomerRemote)i.next();
        String id = (String)customer.getPrimaryKey();
        customerIds.add(id);
         }

      } catch (Exception ex) {
          throw new EJBException("Exception in loadCustomerIds: " +
              ex.getMessage());
      }
   }
```

If a customer's sales representative changes, the client program updates the database by calling the setSalesRepId method of the CustomerBean class. The next time a business method of the SalesRepBean class is called, the ejbLoad method invokes loadCustomerIds, which refreshes the customerIds variable. (To ensure that ejbLoad is invoked before each business method, set the transaction attributes of the business methods to Required.) For example, the SalesRepClient program changes the salesRepId for a customer named Mary Jackson as follows:

```
   CustomerRemote mary = customerHome.findByPrimaryKey("987");
   mary.setSalesRepId("543");
```

The salesRepId value 543 identifies a sales representative named Janice Martin. To list all of Janice's customers, the SalesRepClient program invokes the getCustomerIds method, iterates through the ArrayList of identifiers, and locates each CustomerBean entity bean by calling its findByPrimaryKey method:

```
   SalesRepRemote janice = salesHome.findByPrimaryKey("543");
   ArrayList a = janice.getCustomerIds();
   i = a.iterator();

   while (i.hasNext()) {
      String customerId = (String)i.next();
      CustomerRemote customer =
   customerHome.findByPrimaryKey(customerId);
      String name = customer.getName();
      System.out.println(customerId + ": " + name);
   }
```

Running the SalesRepBean Example

1. In the IDE, choose Tools→PointBase Database→Start Local PointBase Database.
2. Create the database tables by running the `create.sql` script.
   a. Make sure that the `appsrv.root` property in your `<INSTALL>/j2eetutorial14/examples/` file points to the location of your local Application Server installation.
   b. In a terminal window, go to this directory:
   ```
   <INSTALL>/j2eetutorial14/examples/ejb/salesrep/
   ```
   c. Type the following command, which runs the `create.sql` script:
   ```
   asant -buildfile create-db.xml
   ```
3. Choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to `<INSTALL>/j2eetutorial14/examples/ejb/salesrep/`, select the `SalesRepClient` directory, and choose Open Project.
4. The project needs to know the location of some JAR files on its classpath and the SalesRep project. Right-click the SalesRepClient project and choose Resolve Reference Problems. Select the `“SalesRep” project could not be found message` and click Resolve. In the file chooser, select either the completed SalesRep project in `<INSTALL>/j2eetutorial14/examples/ejb/salesrep/` or the project you created and click OK.
5. Select the `“appserv-rt.jar” file/folder could not be found` message and click Resolve. Navigate to the `lib` directory in your application server installation, select `appserv-rt.jar`, and click OK. The IDE automatically resolves the location of `j2ee.jar`. Click Close.
6. Right-click the SalesRep project and choose Deploy Project. The IDE builds the project, deploys the EJB module, and registers a JDBC connection pool and database resource for the project.
7. Right-click the SalesRepClient project and choose Run Project. The client should display the following:
   ```
   ...
   customerId = 221
   customerId = 388
   customerId = 456
   customerId = 844

   987: Mary Jackson
   221: Alice Smith
   388: Bill Williamson
   ```

```
456: Joe Smith
844: Buzz Murphy
...
```

Many-to-Many Relationships

In a many-to-many relationship, each entity can be related to multiple occurrences of the other entity. For example, a college course has many students and each student may take several courses. In a database, this relationship is represented by a cross-reference table containing the foreign keys. In Figure 7–4, the cross-reference table is the `enrollment` table. These tables are accessed by the `StudentBean`, `CourseBean`, and `EnrollerBean` classes.

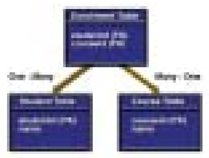


Figure 7–4 Many-to-Many Relationship: Students and Courses

The source code for this example is in this directory:

<INSTALL>`/j2eetutorial14/examples/ejb/enroller/`

To open the project, choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to *<INSTALL>*/j2eetutorial14/examples/ejb/enroller/, select the Enroller directory, and choose Open Project.

The StudentBean and CourseBean classes are complementary. Each class contains an ArrayList of foreign keys. The StudentBean class contains an ArrayList named courseIds, which identifies the courses the student is enrolled in. Similarly, the CourseBean class contains an ArrayList named studentIds.

The ejbLoad method of the StudentBean class adds elements to the courseIds ArrayList by calling loadCourseIds, a private method. The loadCourseIds method gets the course identifiers from the EnrollerBean session bean. The source code for the loadCourseIds method follows:

```
private void loadCourseIds() {

   courseIds.clear();

   try {
      EnrollerRemote enroller = enrollerHome.create();
      ArrayList a = enroller.getCourseIds(studentId);
      courseIds.addAll(a);

  } catch (Exception ex) {
       throw new EJBException("Exception in loadCourseIds: " +
          ex.getMessage());
  }
}
```

Invoked by the loadCourseIds method, the getCourseIds method of the EnrollerBean class queries the enrollment table:

```
select courseid from enrollment
where studentid = ?
```

Only the EnrollerBean class accesses the enrollment table. Therefore, the EnrollerBean class manages the student-course relationship represented in the enrollment table. If a student enrolls in a course, for example, the client calls the enroll business method, which inserts a row:

```
insert into enrollment
values (studentid, courseid)
```

If a student drops a course, the unEnroll method deletes a row:

```
delete from enrollment
where studentid = ? and courseid = ?
```

And if a student leaves the school, the `deleteStudent` method deletes all rows in the table for that student:

```
delete from enrollment
where student = ?
```

The `EnrollerBean` class does not delete the matching row from the `student` table. That action is performed by the `ejbRemove` method of the `StudentBean` class. To ensure that both deletes are executed as a single operation, you must ensure that they belong to the same transaction.

Running the EnrollerBean Example

1. In the IDE, choose Tools→PointBase Database→Start Local PointBase Database.
2. Create the database tables by running the `create.sql` script.
   a. Make sure that the `appsrv.root` property in your `<`*`INSTALL`*`>/j2eetutorial14/examples/` file points to the location of your local Application Server installation.
   b. In a terminal window, go to this directory:
      `<`*`INSTALL`*`>/j2eetutorial14/examples/ejb/enroller/`
   c. Type the following command, which runs the `create.sql` script:
      `asant -buildfile create-db.xml`
3. Choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to `<`*INSTALL*`>/j2eetutorial14/examples/ejb/enroller/`, select the `EnrollerClient` directory, and choose Open Project.
4. The project needs to know the location of some JAR files on its classpath and the Enroller project. Right-click the EnrollerClient project and choose Resolve Reference Problems. Select the `"Enroller" project could not be found message` and click Resolve. In the file chooser, select either the completed Enroller project in `<`*INSTALL*`>/j2eetutorial14/examples/ejb/enroller/` or the project you created and click OK.
5. Select the `"appserv-rt.jar" file/folder could not be found` message and click Resolve. Navigate to the `lib` directory in your application server installation, select `appserv-rt.jar`, and click OK. The IDE automatically resolves the location of `j2ee.jar`. Click Close.

6. Right-click the Enroller project and choose Deploy Project. The IDE builds the project, deploys the EJB module, and registers a JDBC connection pool and database resource for the project.
7. Right-click the EnrollerClient project and choose Run Project. The client should display the following:

```
...
Denise Smith:
220 Power J2EE Programming
333 XML Made Easy
777 An Introduction to Java Programming

An Introduction to Java Programming:
823 Denise Smith
456 Joe Smith
388 Elizabeth Willis
...
```

Primary Keys for Bean-Managed Persistence

You specify the primary key class in the entity bean's deployment descriptor. In most cases, your primary key class will be a `String`, an `Integer`, or some other class that belongs to the J2SE or J2EE standard libraries. For some entity beans, you will need to define your own primary key class. For example, if the bean has a composite primary key (that is, one composed of multiple fields), then you must create a primary key class.

The Primary Key Class

The following primary key class is a composite key, the `productId` and `vendorId` fields together uniquely identify an entity bean.

```
public class ItemKey implements java.io.Serializable {

   public String productId;
   public String vendorId;

   public ItemKey() { };

   public ItemKey(String productId, String vendorId) {
     this.productId = productId;
     this.vendorId = vendorId;
   }

   public String getProductId() {

      return productId;
   }

   public String getVendorId() {

      return vendorId;
   }

   public boolean equals(Object other) {

      if (other instanceof ItemKey) {
         return (productId.equals(((ItemKey)other).productId)
               && vendorId.equals(((ItemKey)other).vendorId));
      }
      return false;
   }

   public int hashCode() {

      return productId.concat(vendorId).hashCode();
   }
}
```

For bean-managed persistence, a primary key class must meet these requirements:

- The access control modifier of the class must be `public`.
- All fields must be declared as `public`.
- The class must have a public default constructor.
- The class must implement the `hashCode()` and `equals(Object other)` methods.
- The class must be serializable.

Primary Keys in the Entity Bean Class

With bean-managed persistence, the `ejbCreate` method assigns the input parameters to instance variables and then returns the primary key class:

```
public ItemKey ejbCreate(String productId, String vendorId,
   String description) throws CreateException {

   if (productId == null || vendorId == null) {
      throw new CreateException(
                "The productId and vendorId are required.");
   }

   this.productId = productId;
   this.vendorId = vendorId;
   this.description = description;

   return new ItemKey(productId, vendorId);
}
```

The `ejbFindByPrimaryKey` verifies the existence of the database row for the given primary key:

```
public ItemKey ejbFindByPrimaryKey(ItemKey primaryKey)
   throws FinderException {

   try {
      if (selectByPrimaryKey(primaryKey))
         return primaryKey;
   ...
}

private boolean selectByPrimaryKey(ItemKey primaryKey)
   throws SQLException {

   String selectStatement =
         "select productid " +
         "from item where productid = ? and vendorid = ?";
   PreparedStatement prepStmt =
         con.prepareStatement(selectStatement);
   prepStmt.setString(1, primaryKey.getProductId());
   prepStmt.setString(2, primaryKey.getVendorId());
   ResultSet rs = prepStmt.executeQuery();
   boolean result = rs.next();
   prepStmt.close();
   return result;
}
```

Getting the Primary Key

A client can fetch the primary key of an entity bean by invoking the `getPrimaryKey` method of the `EJBObject` class:

```
SavingsAccountRemote account;
...
String id = (String)account.getPrimaryKey();
```

The entity bean retrieves its own primary key by calling the `getPrimaryKey` method of the `EntityContext` class:

```
EntityContext context;
...
String id = (String) context.getPrimaryKey();
```

8 Container-Managed Persistence Examples

AN entity bean with container-managed persistence (CMP) offers important advantages to the bean developer. First, the EJB container handles all database storage and retrieval calls. Second, the container manages the relationships between the entity beans. Because of these services, you don't have to code the database access calls in the entity bean. Instead, you specify settings in the bean's deployment descriptor. Not only does this approach save you time, but also it makes the bean portable across various database servers.

This chapter focuses on the source code and deployment settings for an example called `Roster`, an application that features entity beans with container-managed persistence. If you are unfamiliar with the terms and concepts mentioned in this chapter, please consult the section Container-Managed Persistence (page 113).

Overview of the Roster Module

The `Roster` module maintains the team rosters for players in sports leagues. The example has five components. The `RosterClient` component is an application client that accesses the `RosterBean` session bean through the bean's remote interfaces. `RosterBean` accesses three entity beans—`PlayerBean`, `TeamBean`, and `LeagueBean`—through their local interfaces.

The entity beans use container-managed persistence and relationships. The `TeamBean` and `PlayerBean` entity beans have a bidirectional, many-to-many relationship. In a bidirectional relationship, each bean has a relationship field whose value identifies the related bean instance. The multiplicity of the `TeamBean-PlayerBean` relationship is many-to-many: Players who participate in more than one sport belong to multiple teams, and each team has multiple players. The `LeagueBean` and `TeamBean` entity beans also have a bidirectional relationship, but the multiplicity is one-to-many: A league has many teams, but a team can belong to only one league.

Figure 8–1 shows the components and relationships of the `Roster` module. The dotted lines represent the access gained through invocations of the JNDI `lookup` method. The solid lines represent the container-managed relationships.

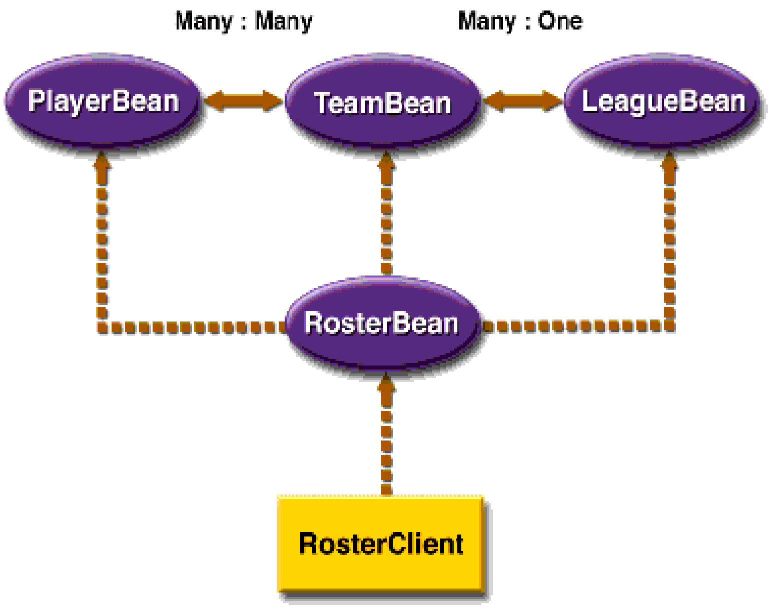


Figure 8–1 `Roster` Example

Creating the Roster EJB Module

To create this project in the IDE, you will create an EJB module project, create the database in PointBase, and generate the CMP entity beans from the database.

You will then create a session bean through which the client application accesses the entity beans.

Complete versions of both the EJB module and the client application for this example are in the <*INSTALL*>`/j2eetutorial14/examples/ejb/cmproster` directory.

Creating the Project

1. Choose File→New Project (Ctrl-Shift-N).
2. From the Enterprise template category, select EJB Module and click Next.
3. Name the project `Roster`, specify a location for the project, and click Finish.

Creating the Database Tables

The `Roster` example uses the database tables shown in Table 8–2.

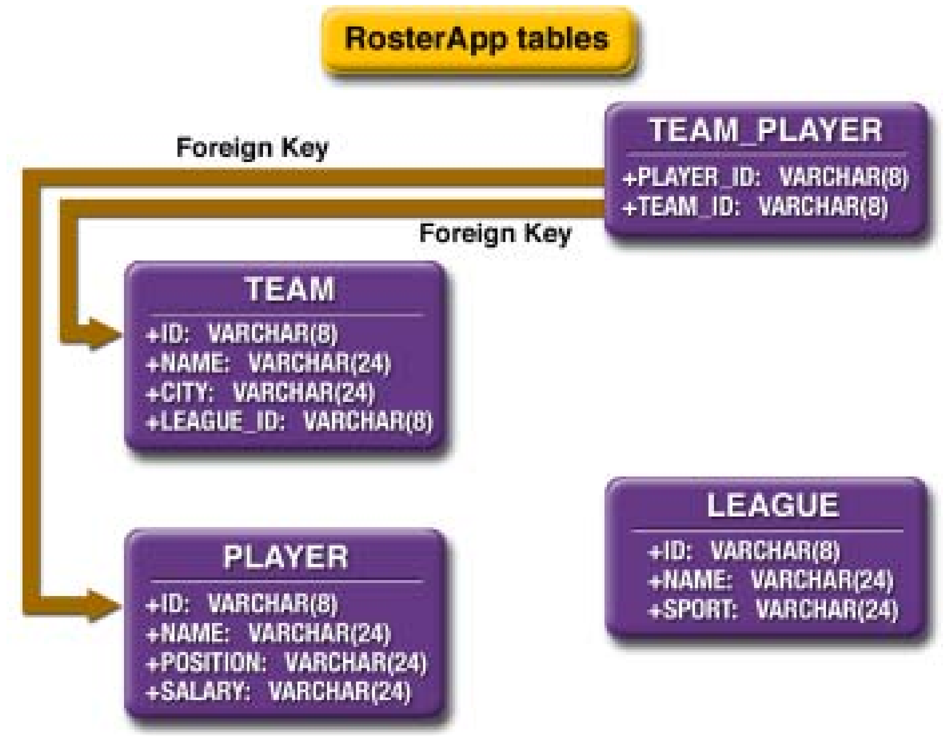


Figure 8–2 Database Tables in `Roster`

The instructions that follow explain how to use the `Roster` example with PointBase, the database software that is included in the Application Server bundle.

1. Choose Tools→PointBase Database→Start Local PointBase Database.
2. Create the database tables by running the `create.sql` script.
   a. Make sure that the `appsrv.root` property in your `<INSTALL>/j2eetutorial14/examples/` file points to the location of your local Application Server installation.
   b. In a terminal window, go to this directory:
   ```
   <INSTALL>/j2eetutorial14/examples/ejb/cmproster/
   ```
   c. Type the following command, which runs the `create.sql` script:
   ```
   asant -buildfile create-db.xml
   ```
3. In the Runtime window, expand the Databases node, right-click the `jdbc:pointbase:server://localhost:9092/sun-appserv-samples` node, and choose Connect. Type `pbpublic` as the password and click OK.

Once the connection is established, expand the connection node's Tables node. There should be nodes for the following tables:

- `LEAGUE`
- `PLAYER`
- `TEAM`
- `TEAM-PLAYER`

Generating the CMP Entity Beans

1. In the Projects window, right-click the Enterprise Beans node for the Roster project and choose New→CMP Entity Beans From Database.
2. In the JDBC Connection combo box, select `jdbc:pointbase://localhost:9092/`sun-appserv-samples. In the Package field, type `team`. Leave the default settings in the rest of the wizard and click Next.
3. From the list, select `PLAYER`, `LEAGUE`, `TEAM`, and `TEAM_PLAYER` and click Add. Then click Finish. You can view the generated entity beans under the project's Enterprise Beans node.

The PlayerBean Code

The `PlayerBean` entity bean represents a player in a sports league. Like any local entity bean with container-managed persistence, `PlayerBean` needs the following code:

- Entity bean class (`PlayerBean`)
- Local home interface (`PlayerLocalHome`)
- Local interface (`PlayerLocal`)

In addition to these standard files, the IDE also creates a business interface (`PlayerLocalBusiness`) in which it registers business methods.

The source code for this example is in the <*INSTALL*>`/j2eetutorial14/examples/ejb/cmproster/Roster/src/java` directory.

Entity Bean Class

The code of the entity bean class must meet the container-managed persistence syntax requirements. First, the class must be defined as `public` and `abstract`. Second, the class must implement the following:

- The `EntityBean` interface
- Zero or more `ejbCreate` and `ejbPostCreate` methods
- The `get` and `set` access methods, defined as `abstract`, for the persistent and relationship fields
- Any select methods, defining them as `abstract`
- The home methods
- The business methods

The entity bean class must not implement these methods:

- The finder methods
- The `finalize` method

Differences between Container-Managed and Bean-Managed Code

Because it contains no calls to access the database, an entity bean with container-managed persistence requires a lot less code than one with bean-managed persistence. For example, the `PlayerBean.java` source file discussed in this chapter is much smaller than the `SavingsAccountBean.java` code documented in Chapter 7. Table 8–1 compares the code of the two types of entity beans.

Table 8–1 Coding Differences between Persistent Types

Difference	**Container-Managed**	**Bean-Managed**
Class definition	Abstract	Not abstract
Database access calls	Handled by container	Coded by developers
Persistent state	Represented by virtual persistent fields	Coded as instance variables
Access methods for persistent and relationship fields	Required	None

Table 8–1 Coding Differences between Persistent Types (Continued)

Difference	**Container-Managed**	**Bean-Managed**
findByPrimaryKey method	Handled by container	Coded by developers
Customized finder methods	Handled by container, but the developer must define the EJB QL queries	Coded by developers
Select methods	Handled by container	None
Return value of ejbCreate	null	Must be the primary key

Note that for both types of persistence, the rules for implementing business and home methods are the same. See the sections The Business Methods (page 180) and The Home Methods (page 182) in Chapter 7.

Access Methods

An entity bean with container-managed persistence has persistent and relationship fields. These fields are virtual, so you do not code them in the class as instance variables. Instead, you specify them in the bean's deployment descriptor. To permit access to the fields, you define abstract `get` and `set` methods in the entity bean class.

Access Methods for Persistent Fields

The EJB container automatically performs the database storage and retrieval of the bean's persistent fields. The deployment descriptor of `PlayerBean` specifies the following persistent fields:

- `id` (primary key)
- `name`
- `position`
- `salary`

You can view the CMP fields for each bean by expanding the project's Configuration Files node, double-clicking `ejb-jar.xml`, and expanding the CMP Fields section for the bean.

The `PlayerBean` class defines the access methods for the persistent fields as follows:

```
public abstract String getId();
public abstract void setId(String id);

public abstract String getName();
public abstract void setName(String name);

public abstract String getPosition();
public abstract void setPosition(String position);

public abstract Double getSalary();
public abstract void setSalary(Double salary);
```

The IDE generates each of these getter and setter methods based on the information it finds in the database. The name of an access method begins with `get` or `set`, followed by the capitalized name of the persistent or relationship field. For example, the accessor methods for the `salary` field are `getSalary` and `setSalary`. This naming convention is similar to that of JavaBeans components.

Access Methods for Relationship Fields

In the `Roster` module, a player can belong to multiple teams, so a `PlayerBean` instance may be related to many `TeamBean` instances. To specify this relationship, the deployment descriptor of `PlayerBean` defines a relationship field named `teams`.

The IDE generates the names of CMP fields and relationship fields based solely on the names of the columns it finds in the database. You can give these fields better names by editing the `ejb-jar.xml` deployment descriptor.

1. In the Projects window, expand the Configuration Files node for the Roster project and double-click `ejb-jar.xml`.
2. In the top of the visual editor, click CMP Relationships. The table lists all of the CMP relationships for the EJB module.
3. Select the `TeamPlayer` row in the table and click Edit. The information on the left of the dialog box defines the `PlayerBean` side of the relationship, while the information on the right describes the `TeamBean` side of the relationship. The multiplicity is set to Many To Many because a team has many players and a player can belong to more than one team.

Under the `PlayerID` role, change the value of the Field Name setting from `teamId` to `teams`. Under the `TeamID` role, change the Field Name setting from `playerId` to `players`. Then click OK.

4. In the `TeamBean-LeagueBean` CMP relationship, change the field name for the `TeamBean` role to `league` and the field name for the `LeagueID` role to `teams`.

Notice that the access methods in the enterprise beans are automatically updated to use the new field names. For example, in the `PlayerBean` class, the access methods for the `teams` relationship field are as follows:

```
public abstract Collection getTeams();
public abstract void setTeams(Collection teams);
```

Finder and Select Methods

Finder and select methods use EJB QL queries to return objects and state information of entity beans using container-managed persistence.

A select method is similar to a finder method in the following ways:

- A select method can return a local or remote interface (or a collection of interfaces).
- A select method queries a database.
- The deployment descriptor specifies an EJB QL query for a select method.
- The entity bean class does not implement the select method.

However, a select method differs significantly from a finder method:

- A select method can return a persistent field (or a collection thereof) of a related entity bean. A finder method can return only a local or remote interface (or a collection of interfaces).
- Because it is not exposed in any of the local or remote interfaces, a select method cannot be invoked by a client. It can be invoked only by the methods implemented within the entity bean class. A select method is usually invoked by either a business or a home method.
- A select method is defined in the entity bean class. For bean-managed persistence, a finder method is defined in the entity bean class, but for container-managed persistence it is not.

The IDE automatically generated finder methods for each of your CMP fields when it generated them from the database. In order to run more sophisticated queries on the database, you have to add some additional finder methods to the `PlayerBean` entity bean.

1. In the Projects window, expand the Configuration Files node and double-click `ejb-jar.xml`.
2. Expand the PlayerEB section and the CMP Finder Methods section and click Add.
3. Use the dialog box to add the finder methods in the following table:

Table 8–2 Additional finder methods for PlayerBean entity bean

Name	**Cardinality**	**EJBQL**	**Parameters**
findAll	Many	`select object(p) from Player p`	`none`
findByCity	Many	`select distinct object(p) from Player p, in (p.teams) as t where t.city = ?1`	`String city`
findByHigherSalary	Many	`select distinct object(p1) from Player p1, Player p2 where p1.salary > p2.sal-ary and p2.name = ?1`	`String name`
findByLeague	Many	`select distinct object(p) from Player p, in (p.teams) as t where t.league = ?1`	`team.League Local league`
findByPositionAnd-Name	Many	`select distinct object(p) from Player p where p.position = ?1 and p.name = ?2`	`String position, String name`
findBySalaryRange	Many	`select distinct object(p) from Player p where p.salary between ?1 and ?2`	`double low, double high`

Table 8–2 Additional finder methods for PlayerBean entity bean

Name	Cardinality	EJBQL	Parameters
findBySport	Many	`select distinct object(p) from Player p, in (p.teams) as t where t.league.sport = ?1`	`String sport`
`findByTest`	Many	`select distinct object(p) from Player p where p.name = ?1`	`String parm1, String parm2, String parm3`
`findNotOnTeam`	Many	`select object(p) from Player p where p.teams is empty`	`none`

You also have to code your bean's select methods.

1. In the PlayerEB section of the `ejb-jar.xml` editor, expand the CMP Select Methods section and click Add.
2. Use the Add Select Method dialog box to add the select methods in the following table:

Table 8–3 Select methods for the PlayerBean entity bean

Name	Return Type	EJBQL	Parameters
ejbSelectLeagues	`java.util.Collection`	`select distinct t.league from Player p, in (p.teams) as t where p = ?1`	`team.PlayerLocal player`
ejbSelectSports	`java.util.Collection`	select distinct t.league.sport from Player p, in (p.teams) as t where p = ?1	`team.PlayerLocal player`

The signature for a select method must follow these rules:

- The prefix of the method name must be `ejbSelect`.
- The access control modifier must be `public`.
- The method must be declared as `abstract`.
- The `throws` clause must include the `javax.ejb.FinderException`.

Helper Classes

The enterprise beans in the Roster EJB module depend on a few helper classes. You have to add these classes to the EJB module before you continue coding the enterprise beans.

1. Copy the *<INSTALL>*`/j2eetutorial14/examples/ejb/cmproster/Roster/src/java/util` directory to the `src/java` directory of your Roster project.
2. In the Projects window, expand the Source Packages node for the Roster project. The `util` package should appear under the node.

Business Methods

Because clients cannot invoke select methods, the `PlayerBean` class wraps them in the `getLeagues` and `getSports` business methods.

Note: You can quickly copy business methods to an enterprise bean by copying and pasting the methods into your bean class, then right-clicking the method name in the Source Editor and choosing EJB Methods→Add to Local Interface. When you are done, press Alt-Shift-F to generate any missing import statements.

1. In the Source Editor, right-click anywhere in the body of the `PlayerBean` class and choose EJB Methods→Add Business Method.
2. In the Name field, type getLeagues. In the Return Type combo box, type Collection.
3. In the Exceptions tab, use the Add button to add a FinderException.
4. Click OK to generate the finder method in both the bean class and the local business interface.
5. Edit the `getLeagues` method as follows:

```
public Collection getLeagues() throws FinderException {

   PlayerLocal player =
      (PlayerLocal)context.getEJBLocalObject();
   return ejbSelectLeagues(player);
}
```

6. Repeat steps 1-5 to enter the following business method:

```
public Collection getSports() throws FinderException {

   PlayerLocal player =
      (team.PlayerLocal)context.getEJBLocalObject();
   return ejbSelectSports(player);
}
```

7. You also have to add a few business methods that manage the contents of each entity bean. Add the following business methods to `TeamBean.java`:

```
public void addPlayer(PlayerLocal player) {
   Debug.print("TeamBean addPlayer");
   try {
      Collection players = getPlayers();
      players.add(player);
   } catch (Exception ex) {
      throw new EJBException(ex.getMessage());
   }
}

public void dropPlayer(PlayerLocal player) {
   Debug.print("TeamBean dropPlayer");
   try {
      Collection players = getPlayers();
      players.remove(player);
   } catch (Exception ex) {
      throw new EJBException(ex.getMessage());
   }
}

public ArrayList getCopyOfPlayers() {
   Debug.print("TeamBean getCopyOfPlayers");
   ArrayList playerList = new ArrayList();
   Collection players = getPlayers();
   Iterator i = players.iterator();
   while (i.hasNext()) {
      PlayerLocal player = (PlayerLocal) i.next();
      PlayerDetails details =
      new   PlayerDetails(player.getId(),   player.getName(),
player.getPosition(), 0.0);
      playerList.add(details);
   }
   return playerList;
}
```

8. Press Alt-Shift-F to generate the following import statements:

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import util.Debug;
import util.PlayerDetails;
```

9. Add the following business methods to `LeagueBean.java`:

```
public void addTeam(team.TeamLocal team) {
   Debug.print("TeamBean addTeam");
   try {
      Collection teams = getTeams();
      teams.add(team);
   } catch (Exception ex) {
      throw new EJBException(ex.getMessage());
   }
}

public void dropTeam(team.TeamLocal team) {
   Debug.print("TeamBean dropTeam");
   try {
      Collection teams = getTeams();
      teams.remove(team);
   } catch (Exception ex) {
      throw new EJBException(ex.getMessage());
   }
}
```

10.Press Alt-Shift-F to generate the following import statements:

```
import java.util.Collection;
import util.Debug;
```

Note: Since you used simple names in the business method declarations, you also need to create import statements in the local business interfaces for the `TeamBean` and `PlayerBean` entity beans. You can quickly open the local business interface by Ctrl-clicking its name in the Source Editor. For example, go to the class declaration of `PlayerBean.java`, hold down the Ctrl key, and click `PlayerLocal-Business`. The class opens in the Source Editor. Then press Alt-Shift-F to generate the import statements.

Entity Bean Methods

Because the container handles persistence, the life-cycle methods in the `Player-Bean` class are nearly empty.

The `ejbCreate` method is generated for you by the IDE. It initializes the bean instance by assigning the input arguments to the persistent fields. At the end of the transaction that contains the create call, the container inserts a row into the database. Here is the source code for the `ejbCreate` method:

```
public String ejbCreate (String id, String name,
    String position, double salary) throws CreateException {

    setPlayerId(id);
    setName(name);
    setPosition(position);
    setSalary(salary);
    return null;
}
```

The `ejbPostCreate` method returns `void`, and it has the same input parameters as the `ejbCreate` method. If you want to set a relationship field to initialize the bean instance, you should do so in the `ejbPostCreate` method. You cannot set a relationship field in the `ejbCreate` method.

Except for a debug statement, the `ejbRemove` method in the `PlayerBean` class is empty. The container invokes `ejbRemove` before removing the entity object.

The container automatically synchronizes the state of the entity bean with the database. After the container loads the bean's state from the database, it invokes the `ejbLoad` method. In like manner, before storing the state in the database, the container invokes the `ejbStore` method.

Refactoring Entity Bean Methods

Refactoring is the process of making application-wide changes to your code without breaking the application's functionality. For example, the `TeamBean` enterprise bean's `ejbCreate` method takes four parameters, including a `League-Local leagueId` object. Change the method signature to remove the `LeagueLo-cal leagueId` object from the list of parameters.

1. In the `ejbCreate` method, delete the following lines:

```
if (leagueId == null) {
  throw     new     javax.ejb.CreateException("The     field
\"leagueId\" must not be null");
}
```

2. In the `ejbPostCreate` method, delete the following line:

   ```
   setLeagueId(leagueId);
   ```

3. Right-click the method name for `ejbCreate` and choose Refactor→Change Method Parameters. Click Next, select `leagueId`, and click Remove. Then click Next to preview the changes that will be made to your code. Notice that the refactoring will change the method signature of both the `ejbCreate` method in the bean class and the `create` method in the home interface.
4. In the Refactoring window, click Do Refactoring.

Local Home Interface

The local home interface defines the `create`, finder, and home methods that can be invoked by local clients.

The syntax rules for a `create` method follow:

- The name must begin with `create`.
- It must have the same number and types of arguments as its matching `ejbCreate` method in the entity bean class.
- It must return the local interface type of the entity bean.
- The `throws` clause must include the exceptions specified by the `throws` clause of the corresponding `ejbCreate` method.
- The `throws` clause must contain the `javax.ejb.CreateException`.

These rules apply for a finder method:

- The name must begin with `find`.
- The return type must be the entity bean's local interface type or a collection of those types.
- The `throws` clause must contain the `javax.ejb.FinderException`.
- The `findByPrimaryKey` method must be defined.

An excerpt of the `PlayerLocalHome` interface follows.

```
package team;

import java.util.*;
import javax.ejb.*;
```

```
public interface PlayerLocalHome extends EJBLocalHome {

    public PlayerLocal create (String id, String name,
        String position, Double salary)
        throws CreateException;

    public PlayerLocal findByPrimaryKey (String key)
        throws FinderException;

    public Collection findByPosition(String position)
        throws FinderException;
     ...
    public Collection findByLeague(LeagueLocal league)
        throws FinderException;
    ...
  }
```

Local Interface

This interface defines the business and access methods that a local client can invoke. When you create enterprise beans in the IDE, the business method signatures are automatically generated to a `LocalBusiness` or `RemoteBusiness` interface that is extended by the bean interface and implemented by the bean class. The advantage of this approach is that it lets you separate the business logic from implementation logic, and that it lets you check at compile-time that your bean implements the given interfaces.

The local interface is therefore almost empty:

```
package team;

import javax.ejb.*;

public interface PlayerLocal extends EJBLocalObject,
PlayerLocalBusiness {

}
```

The `PlayerBean` class implements two business methods: `getLeagues` and `getSports`. It also defines several `get` and `set` access methods for the persistent and relationship fields. The IDE automatically adds both the set and get methods for the fields to the local business interface. An excerpt of the local business method is as follows:

```
package team;

import java.util.Collection;
import javax.ejb.FinderException;

public interface PlayerLocalBusiness {

  public abstract String getId();
  public abstract String getName();
  public abstract void setName(String name);
  public abstract String getPosition();

  ...

  Collection getLeagues() throws FinderException;
  Collection getSports() throws FinderException;

  ...

}
```

Creating the RosterBean Session Bean

You should never directly access entity beans from a client. Instead, clients should access entity beans through the business methods of a a facade session bean. In our example, the `RosterBean` session bean performs this purpose. The source code for the components is in the <*INSTALL*>`/j2eetutorial14/examples/ejb/cmproster` directory.

1. In the Projects window, right-click the Roster node and choose New→Session Bean. Enter `Roster` for the EJB Name, `roster` for the Package Name, and set the bean to generate both remote and local interfaces. Then click Finish.
2. Right-click in the body of the `RosterBean` class and choose Enterprise Resources→Call Enterprise Bean. Select LeagueEB and click OK. Repeat this step to generate lookup code for PlayerEB and TeamEB.
3. Add the following variable declarations to the class:

```
private PlayerLocalHome playerHome = null;
private TeamLocalHome teamHome = null;
private LeagueLocalHome leagueHome = null;
```

4. Change the `ejbCreate`, `ejbActivate`, and `ejbPassivate` methods to get and release bean references. The `ejbActivate` and `ejbPassivate` methods are hidden in the `EJB Infrastructure methods` code fold.

```
public void ejbCreate() {
   Debug.print("RosterBean ejbCreate");
   playerHome = lookupPlayerBean();
   teamHome = lookupTeamBean();
   leagueHome = lookupLeagueBean();
}

public void ejbActivate() {
   Debug.print("RosterBean ejbCreate");
   playerHome = lookupPlayerBean();
   teamHome = lookupTeamBean();
   leagueHome = lookupLeagueBean();
}

public void ejbPassivate() {
   playerHome = null;
   teamHome = null;
   leagueHome = null;
}
```

5. Create the business methods for accessing the entity beans. You can copy the business methods from the `RosterBean` class in the <*INSTALL*>`/j2eetutorial14/examples/ejb/cmproster/Roster/src/java` directory. The business methods start with `testFinder` on line 114 and end with `copyPlayersToDetails` on line 535. You must also overwrite your project's `RosterRemoteBusiness` interface with the contents of the `RosterRemoteBusiness` in <*INSTALL*>`/j2eetutorial14/examples/ejb/cmproster/Roster/src/java`.

6. Select `RosterBean.java` tab in the Source Editor and press Alt-Shift-F to generate the following import statements:

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import javax.ejb.*;
import team.LeagueLocal;
import team.LeagueLocalHome;
import team.PlayerLocal;
import team.PlayerLocalHome;
import team.TeamLocal;
import team.TeamLocalHome;
import util.Debug;
import util.LeagueDetails;
import util.PlayerDetails;
import util.TeamDetails;
```

Method Invocations in the Roster Module

To show how the various components interact, this section describes the sequence of method invocations that occur for particular functions.

The source code for the RosterClient project is in the <*INSTALL*>`/j2eetutorial14/examples/ejb/cmproster` directory. When you open the project, you have to resolve the references to libraries on the project's classpath.

1. Choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to <*INSTALL*>`/j2eetutorial14/examples/ejb/cmproster/`, select the `RosterClient` directory, and choose Open Project.
2. The project needs to know the location of some JAR files on its classpath and the Roster project. Right-click the RosterClient project and choose Resolve Reference Problems. Select the `"Roster" project could not be found message` and click Resolve. In the file chooser, select either the completed Roster project in <*INSTALL*>`/j2eetutorial14/examples/ejb/cmproster/` or the project you created and click OK.
3. Select the `"appserv-rt.jar" file/folder could not be found` message and click Resolve. Navigate to the `lib` directory in your application server installation, select `appserv-rt.jar`, and click OK. The IDE automatically resolves the location of `j2ee.jar`. Click Close.

Creating a Player

1. RosterClient

The `RosterClient` invokes the `createPlayer` business method of the `RosterBean` session bean to create a new player. In the following line of code, the type of the `myRoster` object is `Roster`, the remote interface of `RosterBean`. The argument of the `createPlayer` method is a `PlayerDetails` object, which encapsulates information about a particular player.

```
myRoster.createPlayer(new PlayerDetails("P1", "Phil Jones",
  "goalkeeper", 100.00));
```

2. RosterBean

The createPlayer method of the RosterBean session bean creates a new instance of the PlayerBean entity bean. Because the access of PlayerBean is local, the create method is defined in the local home interface, PlayerLocal-Home. The type of the playerHome object is PlayerLocalHome. Here is the source code for the createPlayer method:

```
public void createPlayer(PlayerDetails details) {

try {
  PlayerLocal player = playerHome.create(details.getId(),
     details.getName(), details.getPosition(),
        new Double(details.getSalary()));
} catch (Exception ex) {
     throw new EJBException(ex.getMessage());
  }
}
```

3. PlayerBean

The ejbCreate method assigns the input arguments to the bean's persistent fields by calling the set access methods. At the end of the transaction that contains the create call, the container saves the persistent fields in the database by issuing an SQL INSERT statement. The code for the ejbCreate method follows.

```
public String ejbCreate (String id, String name,
  String position, Double salary) throws CreateException {

  setId(id);
  setName(name);
  setPosition(position);
  setSalary(salary);
  return null;
}
```

Adding a Player to a Team

1. RosterClient

The RosterClient calls the addPlayer business method of the RosterBean session bean to add player P1 to team T1. The P1 and T1 parameters are the primary keys of the PlayerBean and TeamBean instances, respectively.

```
myRoster.addPlayer("P1", "T1");
```

2. RosterBean

The addPlayer method performs two steps. First, it calls findByPrimaryKey to locate the PlayerBean and TeamBean instances. Second, it invokes the addPlayer business method of the TeamBean entity bean. Here is the source code for the addPlayer method of the RosterBean session bean:

```
public void addPlayer(String playerId, String teamId) {

   try {
      TeamLocal team = teamHome.findByPrimaryKey(teamId);
      PlayerLocal player =
         playerHome.findByPrimaryKey(playerId);
      team.addPlayer(player);
   } catch (Exception ex) {
      throw new EJBException(ex.getMessage());
   }
}
```

3. TeamBean

The TeamBean entity bean has a relationship field named players, a Collection that represents the players that belong to the team. The access methods for the players relationship field are as follows:

```
public abstract Collection getPlayers();
public abstract void setPlayers(Collection players);
```

The addPlayer method of TeamBean invokes the getPlayers access method to fetch the Collection of related PlayerLocal objects. Next, the addPlayer method invokes the `add` method of the `Collection` interface. Here is the source code for the `addPlayer` method:

```
public void addPlayer(PlayerLocal player) {
   try {
      Collection players = getPlayers();
      players.add(player);
   } catch (Exception ex) {
      throw new EJBException(ex.getMessage());
   }
}
```

Removing a Player

1. RosterClient

To remove player `P4`, the client would invoke the `removePlayer` method of the `RosterBean` session bean:

```
myRoster.removePlayer("P4");
```

2. RosterBean

The `removePlayer` method locates the `PlayerBean` instance by calling `findByPrimaryKey` and then invokes the `remove` method on the instance. This invocation signals the container to delete the row in the database that corresponds to the `PlayerBean` instance. The container also removes the item for this instance from the `players` relationship field in the `TeamBean` entity bean. By this removal, the container automatically updates the `TeamBean`-`PlayerBean` relationship. Here is the `removePlayer` method of the `RosterBean` session bean:

```
public void removePlayer(String playerId) {
   try {
      PlayerLocal player =
         playerHome.findByPrimaryKey(playerId);
      player.remove();
   } catch (Exception ex) {
      throw new EJBException(ex.getMessage());
   }
}
```

Dropping a Player from a Team

1. RosterClient

To drop player P2 from team T1, the client would call the dropPlayer method of the RosterBean session bean:

```
myRoster.dropPlayer("P2", "T1");
```

2. RosterBean

The dropPlayer method retrieves the PlayerBean and TeamBean instances by calling their findByPrimaryKey methods. Next, it invokes the dropPlayer business method of the TeamBean entity bean. The dropPlayer method of the RosterBean session bean follows:

```
public void dropPlayer(String playerId, String teamId) {

   try {
      PlayerLocal player =
         playerHome.findByPrimaryKey(playerId);
      TeamLocal team = teamHome.findByPrimaryKey(teamId);
      team.dropPlayer(player);
   } catch (Exception ex) {
      throw new EJBException(ex.getMessage());
   }
}
```

3. TeamBean

The dropPlayer method updates the TeamBean-PlayerBean relationship. First, the method retrieves the Collection of PlayerLocal objects that correspond to the players relationship field. Next, it drops the target player by calling the remove method of the Collection interface. Here is the dropPlayer method of the TeamBean entity bean:

```
public void dropPlayer(PlayerLocal player) {

   try {
      Collection players = getPlayers();
      players.remove(player);
```

```
    } catch (Exception ex) {
       throw new EJBException(ex.getMessage());
    }
 }
```

Getting the Players of a Team

1. RosterClient

The client can fetch a team's players by calling the `getPlayersOfTeam` method of the `RosterBean` session bean. This method returns an `ArrayList` of `Player-Details` objects. A `PlayerDetail` object contains four variables—`playerId`, `name`, `position`, and `salary`—which are copies of the `PlayerBean` persistent fields. The `RosterClient` calls the `getPlayersOfTeam` method as follows:

```
playerList = myRoster.getPlayersOfTeam("T2");
```

2. RosterBean

The `getPlayersOfTeam` method of the `RosterBean` session bean locates the `TeamLocal` object of the target team by invoking the `findByPrimaryKey` method. Next, the `getPlayersOfTeam` method calls the `getPlayers` method of the `TeamBean` entity bean. Here is the source code for the `getPlayersOfTeam` method:

```
public ArrayList getPlayersOfTeam(String teamId) {

   Collection players = null;

   try {
      TeamLocal team = teamHome.findByPrimaryKey(teamId);
      players = team.getPlayers();
   } catch (Exception ex) {
      throw new EJBException(ex.getMessage());
   }

   return copyPlayersToDetails(players);
}
```

The getPlayersOfTeam method returns the ArrayList of PlayerDetails objects that is generated by the copyPlayersToDetails method:

```
private ArrayList copyPlayersToDetails(Collection players) {

  ArrayList detailsList = new ArrayList();
  Iterator i = players.iterator();

  while (i.hasNext()) {
     PlayerLocal player = (PlayerLocal) i.next();
     PlayerDetails details =
        new PlayerDetails(player.getId(),
           player.getName(), player.getPosition(),
           player.getSalary().doubleValue());
        detailsList.add(details);
  }

  return detailsList;
}
```

3. TeamBean

The getPlayers method of the TeamBean entity bean is an access method of the players relationship field:

```
public abstract Collection getPlayers();
```

This method is exposed to local clients because it is defined in the local interface, TeamLocal:

```
public Collection getPlayers();
```

When invoked by a local client, a get access method returns a reference to the relationship field. If the local client alters the object returned by a get access method, it also alters the value of the relationship field inside the entity bean. For example, a local client of the TeamBean entity bean could drop a player from a team as follows:

```
TeamLocal team = teamHome.findByPrimaryKey(teamId);
Collection players = team.getPlayers();
players.remove(player);
```

If you want to prevent a local client from modifying a relationship field in this manner, you should take the approach described in the next section.

Getting a Copy of a Team's Players

In contrast to the methods discussed in the preceding section, the methods in this section demonstrate the following techniques:

- Filtering the information passed back to the remote client
- Preventing the local client from directly modifying a relationship field

1. RosterClient

If you wanted to hide the salary of a player from a remote client, you would require the client to call the `getPlayersOfTeamCopy` method of the `RosterBean` session bean. Like the `getPlayersOfTeam` method, the `getPlayersOfTeamCopy` method returns an `ArrayList` of `PlayerDetails` objects. However, the objects returned by `getPlayersOfTeamCopy` are different: their `salary` variables have been set to zero. The `RosterClient` calls the `getPlayersOfTeamCopy` method as follows:

```
playerList = myRoster.getPlayersOfTeamCopy("T5");
```

2. RosterBean

Unlike the `getPlayersOfTeam` method, the `getPlayersOfTeamCopy` method does not invoke the `getPlayers` access method that is exposed in the `TeamLocal` interface. Instead, the `getPlayersOfTeamCopy` method retrieves a copy of the player information by invoking the `getCopyOfPlayers` business method that is defined in the `TeamLocal` interface. As a result, the `getPlayersOfTeamCopy` method cannot modify the `players` relationship field of `TeamBean`. Here is the source code for the `getPlayersOfTeamCopy` method of `RosterBean`:

```
public ArrayList getPlayersOfTeamCopy(String teamId) {

  ArrayList playersList = null;

  try {
     TeamLocal team = teamHome.findByPrimaryKey(teamId);
     playersList = team.getCopyOfPlayers();
  } catch (Exception ex) {
     throw new EJBException(ex.getMessage());
```

```
    }

    return playersList;
}
```

3. TeamBean

The `getCopyOfPlayers` method of `TeamBean` returns an `ArrayList` of `PlayerDetails` objects. To create this `ArrayList`, the method iterates through the `Collection` of related `PlayerLocal` objects and copies information to the variables of the `PlayerDetails` objects. The method copies the values of `PlayerBean` persistent fields—except for the `salary` field, which it sets to zero. As a result, a player's salary is hidden from a client that invokes the `getPlayersOfTeamCopy` method. The source code for the `getCopyOfPlayers` method of `TeamBean` follows.

```
public ArrayList getCopyOfPlayers() {

    ArrayList playerList = new ArrayList();
    Collection players = getPlayers();

    Iterator i = players.iterator();
    while (i.hasNext()) {
        PlayerLocal player = (PlayerLocal) i.next();
        PlayerDetails details =
            new PlayerDetails(player.getPlayerId(),
                player.getName(), player.getPosition(), 0.00);
            playerList.add(details);
    }

    return playerList;
}
```

Finding the Players by Position

1. RosterClient

The client starts the procedure by invoking the `getPlayersByPosition` method of the `RosterBean` session bean:

```
playerList = myRoster.getPlayersByPosition("defender");
```

2. RosterBean

The `getPlayersByPosition` method retrieves the `players` list by invoking the `findByPosition` method of the `PlayerBean` entity bean:

```
public ArrayList getPlayersByPosition(String position) {

  Collection players = null;

  try {
     players = playerHome.findByPosition(position);
  } catch (Exception ex) {
     throw new EJBException(ex.getMessage());
  }

  return copyPlayersToDetails(players);
}
```

3. PlayerBean

The `PlayerLocalHome` interface defines the `findByPosition` method:

```
public Collection findByPosition(String position)
  throws FinderException;
```

Because the `PlayerBean` entity bean uses container-managed persistence, the entity bean class (`PlayerBean`) does not implement its finder methods. To specify the queries associated with the finder methods, EJB QL queries must be defined in the bean's deployment descriptor. For example, the `findByPosition` method has this EJB QL query:

```
SELECT DISTINCT OBJECT(p) FROM Player p
WHERE p.position = ?1
```

At runtime, when the container invokes the `findByPosition` method, it will execute the corresponding SQL `SELECT` statement.

For details about configuring the EJB QL in the deployment descriptors, see Finder and Select Methods.

Getting the Sports of a Player

1. RosterClient

The client invokes the `getSportsOfPlayer` method of the `RosterBean` session bean:

```
sportList = myRoster.getSportsOfPlayer("P28");
```

2. RosterBean

The `getSportsOfPlayer` method returns an `ArrayList` of `String` objects that represent the sports of the specified player. It constructs the `ArrayList` from a `Collection` returned by the `getSports` business method of the `PlayerBean` `entity` bean. Here is the source code for the `getSportsOfPlayer` method of the `RosterBean` session bean:

```
public ArrayList getSportsOfPlayer(String playerId) {

  ArrayList sportsList = new ArrayList();
  Collection sports = null;

  try {
     PlayerLocal player =
        playerHome.findByPrimaryKey(playerId);
     sports = player.getSports();
  } catch (Exception ex) {
     throw new EJBException(ex.getMessage());
  }

  Iterator i = sports.iterator();
  while (i.hasNext()) {
     String sport = (String) i.next();
     sportsList.add(sport);
  }
  return sportsList;
}
```

3. PlayerBean

The `getSports` method is a wrapper for the `ejbSelectSports` method. Because the parameter of the `ejbSelectSports` method is of type `PlayerLocal`, the `getSports` method passes along a reference to the entity bean instance. The `PlayerBean` class implements the `getSports` method as follows:

```
public Collection getSports() throws FinderException {

  PlayerLocal player =
     (team.PlayerLocal)context.getEJBLocalObject();
  return ejbSelectSports(player);
}
```

The `PlayerBean` class defines the `ejbSelectSports` method:

```
public abstract Collection ejbSelectSports(PlayerLocal player)
  throws FinderException;
```

The bean's deployment descriptor specifies the following EJB QL query for the `ejbSelectSports` method:

```
SELECT DISTINCT t.league.sport
FROM Player p, IN (p.teams) AS t
WHERE p = ?1
```

Because `PlayerBean` uses container-managed persistence, when the `ejbSelectSports` method is invoked the EJB container will execute its corresponding SQL `SELECT` statement.

Building and Running the Roster Example

Once you have coded all of the enterprise beans in the `Roster` example, you do not have to configure any more deployment descriptors or server resources. The IDE configures all of the necessary settings as you create the source code. You will now build and deploy the module as a stand-alone EJB module, then access it from the RosterClient project.

The RosterClient project and a completed Roster project are located at <*INSTALL*>`/j2eetutorial14/examples/ejb/cmproster/`.

Building and Deploying the EJB Module

You can build and deploy the module in one action.

1. In the Runtime window, expand the Servers node, right-click the node for the Sun Java System Application Server, and choose Start/Stop Server. If the server is stopped, click Start Server in the dialog box.
2. In the Projects window, right-click the Roster project and choose Deploy Project.

The IDE does all of the following:

1. Compiles the EJB module's sources and builds the EJB JAR file. You can view the build output in the project's `build` and `dist` directories in the Files window.
2. Registers the JDBC connection pool and datasource on the server.
3. Undeploys the module if it is already deployed to the server.
4. Deploys the module to the server.

Running the Client Application

To run the client, follow these steps:

1. If you have not already opened the RosterClient project, choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to <*INSTALL*>`/j2eetutorial14/examples/ejb/cmproster/`, select the `RosterClient` directory, and choose Open Project.
2. The project needs to know the location of some JAR files on its classpath and the Roster project. Right-click the RosterClient project and choose Resolve Reference Problems. Select the `"Roster" project could not be found message` and click Resolve. In the file chooser, select either the completed Roster project in <*INSTALL*>`/j2eetutorial14/examples/ejb/cmproster/` or the project you created and click OK.
3. Select the `"appserv-rt.jar" file/folder could not be found` message and click Resolve. Navigate to the `lib` directory in your application server installation, select `appserv-rt.jar`, and click OK. The IDE automatically resolves the location of `j2ee.jar`. Click Close.
4. If the PointBase database server is not running, choose Tools→PointBase Database→Start Local PointBase Database.

5. In the Projects window, right-click the RosterClient project and choose Run Project.
6. In the Output window, the client displays the following output:

```
P7 Rebecca Struthers midfielder 777.0
P6 Ian Carlyle goalkeeper 555.0
P9 Jan Wesley defender 100.0
P10 Terry Smithson midfielder 100.0
P8 Anne Anderson forward 65.0

T2 Gophers Manteca
T5 Crows Orland
T1 Honey Bees Visalia

P2 Alice Smith defender 505.0
P5 Barney Bold defender 100.0
P25 Frank Fletcher defender 399.0
P9 Jan Wesley defender 100.0
P22 Janice Walker defender 857.0

L1 Mountain Soccer
L2 Valley Basketball
```

Primary Keys for Container-Managed Persistence

Sometimes you must implement the class and package it along with the entity bean. For example, if your entity bean requires a composite primary key (which is made up of multiple fields) or if a primary key field is a Java programming language primitive type, then you must provide a customized primary key class.

The Primary Key Class

For container-managed persistence, a primary key class must meet the following requirements:

- The access control modifier of the class must be `public`.
- All fields must be declared as `public`.
- The fields must be a subset of the bean's persistent fields.
- The class must have a public default constructor.
- The class must implement the `hashCode()` and `equals(Object other)` methods.
- The class must be serializable.

In the following example, the `PurchaseOrderKey` class implements a composite key for the `PurchaseOrderBean` entity bean. The key is composed of two fields—`productModel` and `vendorId`—whose names must match two of the persistent fields in the entity bean class.

```
public class PurchaseOrderKey implements java.io.Serializable {

  public String productModel;
  public String vendorId;

  public PurchaseOrderKey() { };

  public boolean equals(Object other) {

     if (other instanceof PurchaseOrderKey) {
        return (productModel.equals(
             ((PurchaseOrderKey)other).productModel) &&
             vendorId.equals(
             ((PurchaseOrderKey)other).vendorId));
     }
     return false;
  }

  public int hashCode() {

     return productModel.concat(vendorId).hashCode();
  }
}
```

Primary Keys in the Entity Bean Class

In the `PurchaseOrderBean` class, the following access methods define the persistent fields (`vendorId` and `productModel`) that make up the primary key:

```
public abstract String getVendorId();
public abstract void setVendorId(String id);

public abstract String getProductModel();
public abstract void setProductModel(String name);
```

The next code sample shows the `ejbCreate` method of the `PurchaseOrderBean` class. The return type of the `ejbCreate` method is the primary key, but the return value is `null`. Although it is not required, the `null` return value is recommended for container-managed persistence. This approach saves overhead because the bean does not have to instantiate the primary key class for the return value.

```
public PurchaseOrderKey ejbCreate (String vendorId,
    String productModel, String productName)
    throws CreateException {

    setVendorId(vendorId);
    setProductModel(productModel);
    setProductName(productName);

    return null;
}
```

Generating Primary Key Values

For some entity beans, the value of a primary key has a meaning for the business entity. For example, in an entity bean that represents a player on a sports team, the primary key might be the player's driver's license number. But for other beans, the key's value is arbitrary, provided that it's unique. With container-managed persistence, these key values can be generated automatically by the EJB container. To take advantage of this feature, an entity bean must meet these requirements:

- In the deployment descriptor, the primary key class must be defined as a `java.lang.Object`. The primary key field is not specified.
- In the home interface, the argument of the `findByPrimaryKey` method must be a `java.lang.Object`.

- In the entity bean class, the return type of the `ejbCreate` method must be a `java.lang.Object`.

In these entity beans, the primary key values are in an internal field that only the EJB container can access. You cannot associate the primary key with a persistent field or any other instance variable. However, you can fetch the bean's primary key by invoking the `getPrimaryKey` method on the bean reference, and you can locate the bean by invoking its `findByPrimaryKey` method.

Advanced CMP Topics: The Order Example

The `Order` application is an advanced CMP example. It contains entity beans that have self-referential relationships, one-to-one relationships, unidirectional relationships, unknown primary keys, primitive primary key types, and composite primary keys.

To open the project, choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to <*INSTALL*>/j2eetutorial14/examples/ejb/cmporder/, select the Order directory, and choose Open Project.

Structure of Order

`Order` is a simple inventory and ordering application for maintaining a catalog of parts and placing an itemized order of those parts. It has entity beans that represent parts, vendors, orders, and line items. These entity beans are accessed using a stateful session bean that holds the business logic of the application. A simple command-line client adds data to the entity beans, manipulates the data, and displays data from the catalog.

The information contained in an order can be divided into different elements. What is the order number? What parts are included in the order? What parts make up that part? Who makes the part? What are the specifications for the part? Are there any schematics for the part? `Order` is a simplified version of an ordering system that has all these elements.

`Order` consists of two modules: `Order`, an enterprise bean JAR file containing the entity beans, the stateful session bean that accesses the data in the entity beans, the support classes, and the database schema file; and `OrderClient`, the application client that populates the entity beans with data and manipulates the data, displaying the results in a terminal.

Figure 8–3 shows `Order`'s database tables.

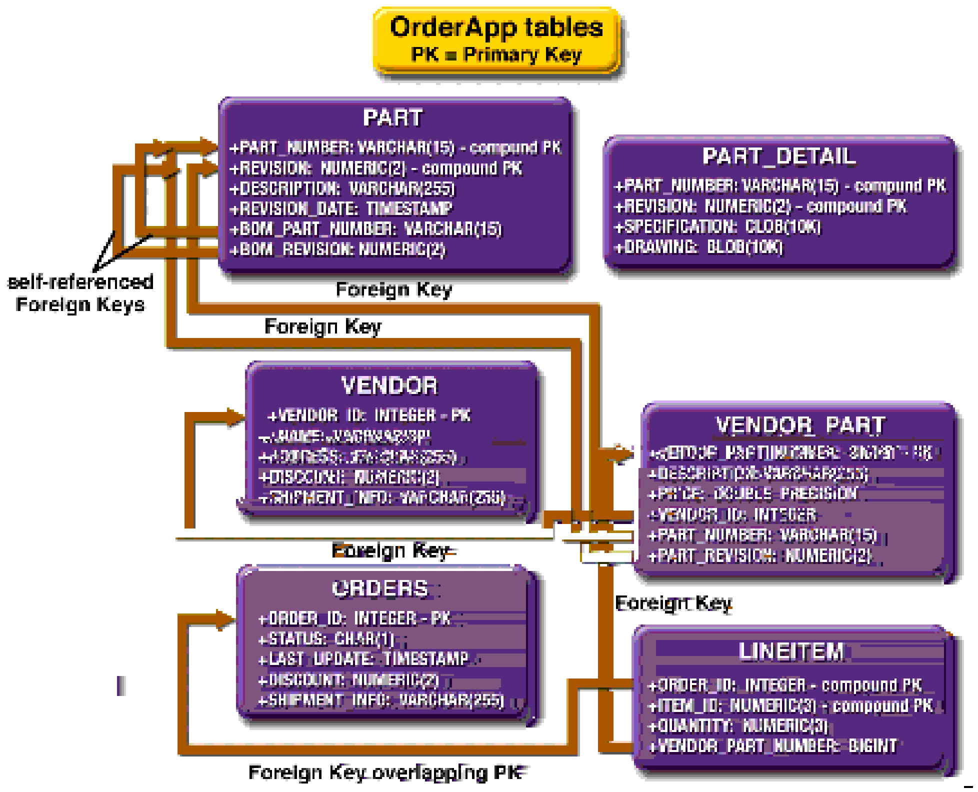

OrderApp tables
PK = Primary Key
PART
+PART_NUMBER: VARCHAR(15) - compund PK
+REVISION: NUMERIC(2) - compound PK
+DESCRIPTION: VARCHAR(255)
+REVISION_DATE: TIMESTAMP
+BOM_PART_NUMBER: VARCHAR(15)
+BOM_REVISION: NUMERIC(2)
PART_DETAIL
+PART_NUMBER: VARCHAR(15) - compund PK
+REVISION: NUMERIC(2) - compound PK
+SPECIFICATION: CLOB(10K)
+DRAWING: BLOB(10K)
self-referenced Foreign Keys
Foreign Key
Foreign Key
VENDOR
+VENDOR_ID: INTEGER - PK
+NAME: VARCHAR(30)
+ADDRESS: VARCHAR(255)
+DISCOUNT: NUMERIC(2)
+SHIPMENT_INFO: VARCHAR(255)
VENDOR_PART
+VENDOR_PART_NUMBER: BIGINT - PK
+DESCRIPTION: VARCHAR(255)
+PRICE: DOUBLE PRECISION
+VENDOR_ID: INTEGER
+PART_NUMBER: VARCHAR(15)
+PART_REVISION: NUMERIC(2)
Foreign Key
ORDERS
+ORDER_ID: INTEGER - PK
+STATUS: CHAR(1)
+LAST_UPDATE: TIMESTAMP
+DISCOUNT: NUMERIC(2)
+SHIPMENT_INFO: VARCHAR(255)
Foreign Key
LINEITEM
+ORDER_ID: INTEGER - compound PK
+ITEM_ID: NUMERIC(3) - compound PK
+QUANTITY: NUMERIC(3)
+VENDOR_PART_NUMBER: BIGINT
Foreign Key overlapping PK

Figure 8–3 Database Tables in `Order`

Bean Relationships in Order

The `Order` example application shows how to set up one-to-many and many-to-many relationships between entity beans. `Order` demonstrates two additional types of entity bean relationships (see Figure 8–4): one-to-one and self-referential relationships.

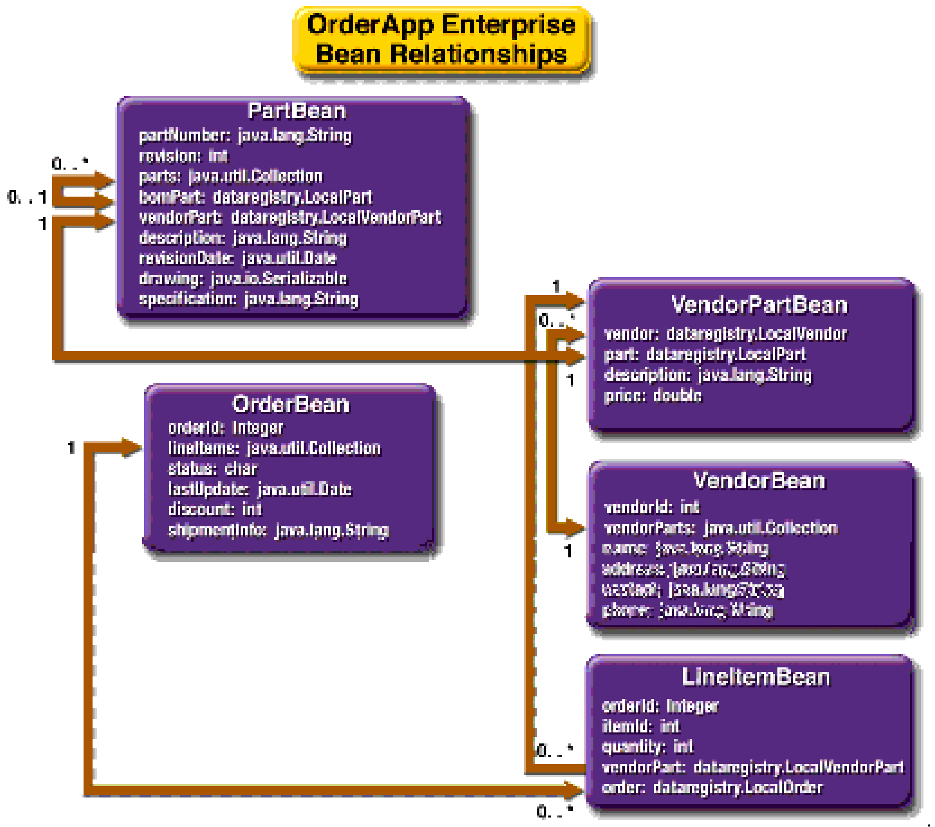

OrderApp Enterprise Bean Relationships
PartBean
partNumber: java.lang.String
revision: int
parts: java.util.Collection
bomPart: dataregistry.LocalPart
vendorPart: dataregistry.LocalVendorPart
description: java.lang.String
revisionDate: java.util.Date
drawing: java.io.Serializable
specification: java.lang.String
VendorPartBean
vendor: dataregistry.LocalVendor
part: dataregistry.LocalPart
description: java.lang.String
price: double
OrderBean
orderId: Integer
lineItems: java.util.Collection
status: char
lastUpdate: java.util.Date
discount: int
shipmentInfo: java.lang.String
VendorBean
vendorId: int
vendorParts: java.util.Collection
name: java.lang.String
address: java.lang.String
contact: java.lang.String
phone: java.lang.String
LineItemBean
orderId: Integer
itemId: int
quantity: int
vendorPart: dataregistry.LocalVendorPart
order: dataregistry.LocalOrder
0..*
0..1
1
1
0..*
1
1
1
0..*
0..*

Figure 8–4 Relationships between Entity Beans in `Order`

Self-Referential Relationships

A *self-referential* relationship is a relationship between container-managed relationship fields (CMR) in the same entity bean. `PartBean` has a CMR field `bomPart` that has a one-to-many relationship with the CMR field `parts`, which is also in `PartBean`. That is, a part can be made up of many parts, and each of those parts has exactly one bill-of-material part.

The primary key for `PartBean` is a compound primary key, a combination of the `partNumber` and `revision` fields. It is mapped to the `PART_NUMBER` and `REVISION` columns in the `PART` table.

One-to-One Relationships

`PartBean` has a CMR field, `vendorPart`, that has a one-to-one relationship with `VendorPartBean`'s CMR field `part`. That is, each part has exactly one vendor part, and vice versa.

One-to-Many Relationship Mapped to Overlapping Primary and Foreign Keys

`OrderBean` has a CMR field, `lineItems`, that has a one-to-many relationship with `LineItemBean`'s CMR field `order`. That is, each order has one or more line item.

`LineItemBean` uses a compound primary key that is made up of the `orderId` and `itemId` fields. This compound primary key maps to the `ORDER_ID` and `ITEM_ID` columns in the `LINEITEM` database table. `ORDER_ID` is a foreign key to the `ORDER_ID` column in the `ORDERS` table. This means that the `ORDER_ID` column is mapped twice: once as a primary key field, `orderId`; and again as a relationship field, `order`.

Unidirectional Relationships

`LineItemBean` has a CMR field, `vendorPart`, that has a unidirectional many-to-one relationship with `VendorPartBean`. That is, there is no CMR field in the target entity bean in this relationship.

Primary Keys in Order's Entity Beans

The `Order` example uses more complicated primary keys than does `Roster`.

Unknown Primary Keys

In `Order`, `VendorPartBean` uses an unknown primary key. That is, the enterprise bean does not specify a primary key field, and uses `java.lang.Object` as the primary key class.

The `LocalVendorPartHome` interface's `findByPrimaryKey` method is defined as follows:

```
public LocalVendorPart findByPrimaryKey(Object aKey)
  throws FinderException;
```

See Generating Primary Key Values (page 243) for more information on unkown primary keys.

Primitive Type Primary Keys

`VendorBean` uses a primary key that is a Java programming language primitive type, an `int`. To use a primitive type as the primary key, you must create a wrapper class. `VendorKey` is the wrapper class for `VendorBean`.

The wrapper primary key class has the same requirements as described in The Primary Key Class (page 242). This is the `VendorKey` wrapper class:

```
package dataregistry;
public final class VendorKey implements java.io.Serializable {

  public int vendorId;

  public boolean equals(Object otherOb) {

    if (this == otherOb) {
      return true;
    }
    if (!(otherOb instanceof VendorKey)) {
      return false;
    }
    VendorKey other = (VendorKey) otherOb;
    return (vendorId == other.vendorId);
  }
  public int hashCode() {
    return vendorId;
  }
  public String toString() {
    return "" + vendorId;
  }
}
```

Compound Primary Keys

A compound primary key is made up of multiple fields and follows the requirements described in The Primary Key Class (page 242). To use a compound primary key, you must create a wrapper class.

In Order, two entity beans use compound primary keys: PartBean and LineItemBean.

PartBean uses the PartKey wrapper class. PartBean's primary key is a combination of the part number and the revision number. PartKey encapsulates this primary key.

LineItemBean uses the LineItemKey class. LineItemBean's primary key is a combination of the order number and the item number. LineItemKey encapsulates this primary key. This is the LineItemKey compound primary key wrapper class:

```
package dataregistry;

public final class LineItemKey implements
      java.io.Serializable {

  public Integer orderId;
  public int itemId;

  public boolean equals(Object otherOb) {
    if (this == otherOb) {
      return true;
    }
    if (!(otherOb instanceof LineItemKey)) {
      return false;
    }
    LineItemKey other = (LineItemKey) otherOb;
    return ((orderId==null?other.orderId==null:orderId.equals
        (other.orderId)) && (itemId == other.itemId));
  }

  public int hashCode() {
    return ((orderId==null?0:orderId.hashCode())
        ^ ((int) itemId));
  }

  public String toString() {
    return "" + orderId + "-" + itemId;
  }
}
```

Entity Bean Mapped to More Than One Database Table

PartBean's fields map to more than one database table: PART and PART_DETAIL. The PART_DETAIL table holds the specification and schematics for the part.

Finder and Selector Methods

VendorBean has two finder methods: findByPartialName and findByOrder. The findByPartialName method searches through the vendor list for matches to a partial name. findByOrder finds all vendors for a particular order.

LineItemBean has one finder method, findAll, which finds all line items.

OrderBean has one selector method, ejbSelectAll, which returns all orders.

VendorPartBean has two selector methods. ejbSelectAvgPrice returns the average price of all parts from a vendor. ejbSelectTotalPricePerVendor returns the price of all the parts from a particular vendor.

Selector methods cannot be accessed outside a bean instance because the selector methods are not defined in the bean interface. If you are using a selector method to return data to a caller, the selector method must be called from a home or business method. In Order, the LocalVendorPartHome.getAvgPrice method returns the result of the ejbSelectAvgPrice method in VendorPartBean.

The return type of a selector query is usually defined by the return type of the ejbSelect methods. You must specify the return type as Remote if the method returns a remote interface or a java.util.Collection of remote interfaces. If the return type is a local interface or a java.util.Collection of local interfaces, set the return type to Local. If the return type is neither a local nor a remote interface, nor a collection of local or remote interfaces, do not set the return type. The OrderBean.ejbSelectAll method returns a collection of local interfaces. VendorPartBean.ejbSelectAvgPrice and VendorPartBean.ejbSelectTotalPricePerVendor return a Double, so the return type is set to None.

Using Home Methods

Home methods are defined in the home interface of a bean and correspond to methods named ejbHome<*METHOD*> in the bean class. For example, a method getValue, defined in the LocalExampleHome interface, corresponds to the ejb-

HomeGetValue method implemented in ExampleBean. The ejbHome<*METHOD*> methods are implemented by the bean developer.

Order uses three home methods: OrderLocalHome.adjustDiscount, VendorPartLocalHome.getAvgPrice, and VendorPartLocalHome.getTotalPricePerVendor. Home methods operate on all instances of a bean rather than on any particular bean instance. That is, home methods cannot access the container-managed fields and relationships of a bean instance on which the method is called.

For example, OrderLocalHome.adjustDiscount is used to increase or decrease the discount on all orders.

Cascade Deletes in Order

Entity beans that use container-managed relationships often have dependencies on the existence of the other bean in the relationship. For example, a line item is part of an order, and if the order is deleted, then the line item should also be deleted. This is called a cascade delete relationship.

In Order, there are two cascade delete dependencies in the bean relationships. If the OrderBean to which a LineItemBean is related is deleted, then the LineItemBean should also be deleted. If the VendorBean to which a VendorPartBean is related is deleted, then the VendorPartBean should also be deleted.

BLOB and CLOB Database Types in Order

The PART_DETAIL table in the database has a column, DRAWING, of type BLOB. BLOB stands for binary large objects, which are used for storing binary data such as an image. The DRAWING column is mapped to the container-managed field PartBean. drawing of type java.io.Serializable.

PART_DETAIL also has a column, SPECIFICATION, of type CLOB. CLOB stands for character large objects, which are used to store string data too large to be stored in a VARCHAR column. SPECIFICATION is mapped to the container-managed field PartBean.specification of type java.lang.String.

Note: You cannot use a BLOB or CLOB column in the WHERE clause of a finder or selector EJB QL query.

Building and Running the Order Example

In order to run the `OrderClient` example, you have to build and deploy the `Order` EJB module and create the database tables.

Building and Deploying the EJB Module

You can build and deploy the module in one action.

1. In the Runtime window, expand the Servers node, right-click the node for the Sun Java System Application Server, and choose Start/Stop Server. If the server is stopped, click Start Server in the dialog box.
2. In the Projects window, right-click the Order project and choose Deploy Project.

The IDE does all of the following:

1. Compiles the EJB module's sources and builds the EJB JAR file. You can view the build output in the project's `build` and `dist` directories in the Files window.
2. Registers the JDBC connection pool and datasource on the server.
3. Undeploys the module if it is already deployed to the server.
4. Deploys the module to the server.

Running the OrderClient Example

1. In the IDE, choose Tools→PointBase Database→Start Local PointBase Database.
2. Create the database tables by running the `create.sql` script.
   a. Make sure that the `appsrv.root` property in your `<`*`INSTALL`*`>/j2eetutorial14/examples/` file points to the location of your local Application Server installation.
   b. In a terminal window, go to this directory:
   ```
   <INSTALL>/j2eetutorial14/examples/ejb/order/
   ```
   c. Type the following command, which runs the `create.sql` script:
   ```
   asant -buildfile create-db.xml
   ```
3. Choose File→Open Project (Ctrl-Shift-O). In the file chooser, go to <*INSTALL*>`/j2eetutorial14/examples/ejb/order/`, select the `OrderClient` directory, and choose Open Project.

4. The project needs to know the location of some JAR files on its classpath and the Enroller project. Right-click the EnrollerClient project and choose Resolve Reference Problems. Select the `“Order” project could not be found message` and click Resolve. In the file chooser, select either the completed Enroller project in <*INSTALL*>`/j2eetutorial14/examples/ejb/enroller/` or the project you created and click OK.
5. Select the `“appserv-rt.jar” file/folder could not be found` message and click Resolve. Navigate to the `lib` directory in your application server installation, select `appserv-rt.jar`, and click OK. The IDE automatically resolves the location of `j2ee.jar`. Click Close.
6. Right-click the OrderClient project and choose Run Project. The client should display the following:

```
Cost of Bill of Material for PN SDFG-ERTY-BN Rev: 7: $241.86
Cost of Order 1111: $664.68
Cost of Order 4312: $2,011.44

Adding 5% discount
Cost of Order 1111: $627.75
Cost of Order 4312: $1,910.87

Removing 7% discount
Cost of Order 1111: $679.45
Cost of Order 4312: $2,011.44

Average price of all parts: $117.55

Total price of parts for Vendor 100: $501.06

Ordered list of vendors for order 1111
200 Gadget, Inc. Mrs. Smith
100 WidgetCorp Mr. Jones

Found 6 line items

Removing Order
Found 3 line items

Found 1 out of 2 vendors with 'I' in the name:
Gadget, Inc.
```

9

A Message-Driven Bean Example

BECAUSE message-driven beans are based on the Java Message Service (JMS) technology, to understand the example in this chapter you should be familiar with basic JMS concepts such as queues and messages.

This chapter describes the source code of a simple message-driven bean example. Before proceeding, you should read the basic conceptual information in the section What Is a Message-Driven Bean? (page 116).

Example Application Overview

The SimpleMessage application has the following components:

- SimpleMessageClient: An application client that sends several messages to a queue
- SimpleMessageMDB: A message-driven bean that asynchronously receives and processes the messages that are sent to the queue

Figure 9–1 illustrates the structure of this application. The application client sends messages to the queue, and the JMS provider (in this case, the Application Server) delivers the messages to the instances of the message-driven bean, which then processes the messages.

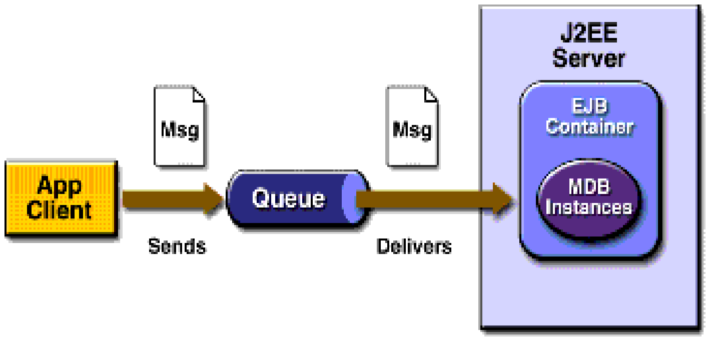


Figure 9–1 The SimpleMessageClient Application

The source code for this application is in the <*INSTALL*>/j2eetutorial14/examples/ejb/simplemessage/ directory.

The SimpleMessageClient Application

The SimpleMessageClient application is a simple Java application that sends messages to a queue. The application locates the connection factory and queue and then generates some messages to send to the queue.

Creating the SimpleMessageClient application

In this example, using the IDE you create the simple Java client application.

1. Choose File→New Project (Ctrl-Shift-N) from the main menu.
2. Select General in the Categories pane and Java Application in the Projects pane and click Next.
3. Enter SimpleMessageClient as the Project Name, specify the project location, and click Finish.

The IDE creates a new project called SimpleMessageClient and the main class opens in the Source Editor. In the Projects window, notice that the main method is located in the Source Packages node in the simplemessageclient package. To run the SimpleMessageClient project you need to add some libraries to the project classpath. You can add the libraries in the Projects window

1. Expand the SimpleMessageClient node, right-click the Libraries node and choose Add JAR/Folder from the contextual menu.
2. In the Add JAR/Folder dialog box, locate and add the following JAR files:
   - j2ee.jar
   - appserv-rt.jar
   - appserv-admin.jar
   - imqjmsra.jar

   With the exception of the imqjmsra.jar file, the JAR files can be found in the lib folder of the local installation of the SJS Application Server. To add the imqjmsra.jar file to the classpath, you first need to extract the JAR file from the imqjmsra.rar file, which is located in the imq/lib folder of the local SJS Application Server installation.
3. Click OK.

After adding the libraries to the classpath, add the following field declarations to the main method in the Source Editor:

```
Context jndiContext = null;
ConnectionFactory connectionFactory = null;
Connection connection = null;
Session session = null;
Destination destination = null;
MessageProducer messageProducer = null;
TextMessage message = null;
final int NUM_MSGS = 3;
```

Press Alt-Shift-F to add and fix any import statements. The import statements should be as follows:

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Queue;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
```

Add the following code to create the new context:

```
try {
    jndiContext = new InitialContext();
} catch (NamingException e) {
    System.out.println("Could not create JNDI " + "context: " +
            e.toString());
    System.exit(1);
}
```

Add the following code for locating the connection factory and queue:

```
try {
    connectionFactory =
        (ConnectionFactory) jndiContext.lookup(
        "jms/SimpleMessageDestinationFactory");
    destination =
        (Queue) jndiContext.lookup("jms/SimpleMessageBean");
} catch (NamingException e) {
    System.out.println("JNDI lookup failed: " + e.toString());
    System.exit(1);
}
```

Add the following code to create the queue connection, session, and sender:

```
try {
    connection = connectionFactory.createConnection();
    session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    messageProducer = session.createProducer(destination);
```

Finally, add the following code to send several messages to the queue and print a message to the server log:

```
    message = session.createTextMessage();

    for (int i = 0; i < NUM_MSGS; i++) {
        message.setText("This is message " + (i + 1));
        System.out.println("Sending message: " +
            message.getText());
        messageProducer.send(message);
    }

    System.out.println("To see if the bean received the messages,");
    System.out.println(
            " check <install_dir>/domains/domain1/logs/server.log.");
} catch (JMSException e) {
    System.out.println("Exception occurred: " + e.toString());
```

```
} finally {
    if (connection != null) {
        try {
            connection.close();
```

The SimpleMessage module appears in the Projects window of the IDE. The next step is to add a message-driven bean to the module.

Creating the SimpleMessageMDB

The code for the SimpleMessageBean class illustrates the requirements of a message-driven bean class:

- It must implement the MessageDrivenBean and MessageListener interfaces.
- The class must be defined as public.
- The class cannot be defined as abstract or final.
- It must implement one onMessage method.
- It must implement one ejbCreate method and one ejbRemove method.
- It must contain a public constructor with no arguments.
- It must not define the finalize method.

Unlike session and entity beans, message-driven beans do not have the remote or local interfaces that define client access. Client components do not locate message-driven beans and invoke methods on them. Although message-driven beans do not have business methods, they may contain helper methods that are invoked internally by the onMessage method.

To create the message-driven bean, perform the following steps:

1. Right-click the SimpleMessage node and choose New→Message-Driven Bean.
2. Enter SimpleMessage as the Ejb Name.
3. Enter beans as the Package name.
4. Select queue as the Destination Type and click Finish.

The IDE creates the SimpleMessage enterprise bean and opens the SimpleMessageBean class in the Source Editor.

For this example, the destination type is being specified as queue. The Destination Type can be either javax.jms.Queue or javax.jms.Topic. A queue uses the point-to-point messaging domain and can have at most one consumer. A topic uses the publish/subscribe messaging domain; it can have zero, one, or many consumers.

When you create the message-driven bean, the IDE generates the following methods in the SimpleMessageBean class:

- ejbCreate
- ejbRemove
- onMessage

These methods are hidden in the code fold in the Source Editor. Expand the code fold to see the methods generated by the IDE and to add the logic to the methods.

The ejbCreate and ejbRemove Methods

The signatures of these methods have the following requirements:

- The access control modifier must be public.
- The return type must be void.
- The modifier cannot be static or final.
- The throws clause must not define any application exceptions.
- The method has no arguments.

In SimpleMessageBean, the ejbCreate and ejbRemove methods are empty. These methods are required, but for this example the methods are not used and are emtpy.

The onMessage Method

When the queue receives a message, the EJB container invokes the onMessage method of the message-driven bean.

The onMessage method is called by the bean's container when a message has arrived for the bean to service. This method contains the business logic that handles the processing of the message. It is the message-driven bean's responsibility to parse the message and perform the necessary business logic.

The onMessage method has a single argument: the incoming message.

The message-driven bean class defines one onMessage method, whose signature must follow these rules:

- The method must be declared as public and must not be declared as final or static.
- The return type must be void.
- The method must have a single argument of type javax.jms.Message.
- The throws clause must not define any application exceptions.
- The onMessage method must be invoked in the scope of a transaction that is determined by the transaction attribute specified in the deployment descriptor.

In the SimpleMessageBean class, the onMessage method casts the incoming message to a TextMessage and displays the text. In the Source Editor, edit the onMessage method as follows:

```
public void onMessage(javax.jms.Message aMessage) {
    TextMessage msg = null;

    try {
        if (aMessage instanceof TextMessage) {
            msg = (TextMessage) aMessage;
            logger.info("MESSAGE BEAN: Message received: " +
                msg.getText());
        } else {
            logger.warning("Message of wrong type: " +
                aMessage.getClass().getName());
        }
    } catch (JMSException e) {
        e.printStackTrace();
        mdc.setRollbackOnly();
    } catch (Throwable te) {
        te.printStackTrace();
    }
}
```

In the Source Editor, add the following code to the public class declaration to print information to the server log:

```
static final Logger logger = Logger.getLogger("SimpleMessageBean");
```

Now import any necessary libraries for the message-driven bean. For the SimpleMessage example, add the following import statements:

```
import javax.jms.JMSException;
import javax.jms.TextMessage;
import java.util.logging.Logger;
```

Import statements can be added manually, or the IDE can check and fix any import statements in the class. To automatically add and fix the import statements, place the insertion point anywhere in the body of the class in the Source Editor and press Alt-Shift-F to Fix Imports. The IDE removes any unused import statements and adds any missing important statements.

You are prompted by a dialog box when the IDE cannot locate a library or there is more than one possible library. When there is more than one possible matching library, select the correct library from the combo box.

Building and Deploying SimpleMessage Module

Now that you have finished creating the EJB module, the next step is to build and deploy the application to the SJS Application Server from within the IDE. The source files for the SimpleMessage example are available in the *<INSTALL>*/j2eetutorial14/examples/ejb/simplemessage directory.

Building and Deploying the Application

After assembling and adding the message-driven bean to the EJB module, you can build and deploy the application.

1. In the Projects window, right-click the SimpleMessage node and select Build Main Project (F11) from the contextual menu.
2. Look at the Output window to ensure the application was built successfully.
3. In the Projects window, right-click the SimpleMessage node and select Deploy Project from the contextual menu.

When you deploy the SimpleMessage example, the IDE registers the JMS resources with the SJS Application Server. To see the registered resources, expand the Servers node in the Runtime window of the IDE and expand the JMS Resources node under the SJS Application Server instance. The IDE also registers the related connector resources. The connector resources are visible in the Connectors node in the Runtime window.

The deployed SimpleMessage application is visible in the Runtime window of the IDE. To see the deployed application, expand the EJB Modules node in the Applications node of the server instance. You can undeploy and disable the application in the Runtime window.

Running the Client

After deploying the SimpleMessage application, run the SimpleMessageClient to send a message to the SimpleMessage application.

1. In the Projects window, right-click the SimpleMessageClient node and select Run Project from the contextual menu.

The following lines are displayed in the Output window of the IDE:

```
Sending message: This is message 1
Sending message: This is message 2
Sending message: This is message 3
To see if the bean received the messages,
 check <install_dir>/domains/domain1/logs/server.log.
```

In the server log file, the following lines should be displayed, wrapped in logging information:

```
MESSAGE BEAN: Message received: This is message 1
MESSAGE BEAN: Message received: This is message 2
MESSAGE BEAN: Message received: This is message 3
```

Undeploy the application after you finish running the client.

Removing the Administered Objects

After you run the example, you can delete the connection factory and queue in the Runtime window of the IDE.

1. Expand the SJS Application Server instance node under Servers in the Runtime window.

2. Expand the JMS Resources node and the Connection Factories and Destination Resources nodes.
3. Right-click the resources and select Delete Resource from the contextual menu.

When you delete the JMS resources, the related connector resources are also deleted. The resources are registered again when you redeploy the SimpleMessage application.

Glossary

abstract schema
The part of an entity bean's deployment descriptor that defines the bean's persistent fields and relationships.

abstract schema name
A logical name that is referenced in EJB QL queries.

access control
The methods by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints.

ACID
The acronym for the four properties guaranteed by transactions: atomicity, consistency, isolation, and durability.

activation
The process of transferring an enterprise bean from secondary storage to memory. (See *passivation*.)

anonymous access
Accessing a resource without authentication.

applet
A J2EE component that typically executes in a web browser but can execute in a variety of other applications or devices that support the applet programming model.

applet container
A container that includes support for the applet programming model.

application assembler
A person who combines J2EE components and modules into deployable application units.

application client
A first-tier J2EE client component that executes in its own Java virtual machine. Application clients have access to some J2EE platform APIs.

application client container
A container that supports application client components.

application client module

A software unit that consists of one or more classes and an application client deployment descriptor.

application component provider

A vendor that provides the Java classes that implement components' methods, JSP page definitions, and any required deployment descriptors.

application configuration resource file

An XML file used to configure resources for a JavaServer Faces application, to define navigation rules for the application, and to register converters, validators, listeners, renderers, and components with the application.

archiving

The process of saving the state of an object and restoring it.

attribute

A qualifier on an XML tag that provides additional information.

authentication

The process that verifies the identity of a user, device, or other entity in a computer system, usually as a prerequisite to allowing access to resources in a system. The Java servlet specification requires three types of authentication—basic, form-based, and mutual—and supports digest authentication.

authorization

The process by which access to a method or resource is determined. Authorization depends on the determination of whether the principal associated with a request through authentication is in a given security role. A security role is a logical grouping of users defined by the person who assembles the application. A deployer maps security roles to security identities. Security identities may be principals or groups in the operational environment.

authorization constraint

An authorization rule that determines who is permitted to access a web resource collection.

B2B

Business-to-business.

backing bean

A JavaBeans component that corresponds to a JSP page that includes JavaServer Faces components. The backing bean defines properties for the components on the page and methods that perform processing for the component. This processing includes event handling, validation, and processing associated with navigation.

basic authentication
An authentication mechanism in which a web server authenticates an entity via a user name and password obtained using the web application's built-in authentication mechanism.

bean-managed persistence
The mechanism whereby data transfer between an entity bean's variables and a resource manager is managed by the entity bean.

bean-managed transaction
A transaction whose boundaries are defined by an enterprise bean.

binary entity
See *unparsed entity.*

binding (XML)
Generating the code needed to process a well-defined portion of XML data.

binding (JavaServer Faces technology)
Wiring UI components to back-end data sources such as backing bean properties.

build file
The XML file that contains one or more asant targets. A target is a set of tasks you want to be executed. When starting asant, you can select which targets you want to have executed. When no target is given, the project's default target is executed.

business logic
The code that implements the functionality of an application. In the Enterprise JavaBeans architecture, this logic is implemented by the methods of an enterprise bean.

business method
A method of an enterprise bean that implements the business logic or rules of an application.

callback methods
Component methods called by the container to notify the component of important events in its life cycle.

caller
Same as *caller principal*.

caller principal
The principal that identifies the invoker of the enterprise bean method.

cascade delete
A deletion that triggers another deletion. A cascade delete can be specified for an entity bean that has container-managed persistence.

CDATA
A predefined XML tag for character data that means “don’t interpret these characters,” as opposed to parsed character data (PCDATA), in which the normal rules of XML syntax apply. CDATA sections are typically used to show examples of XML syntax.

certificate authority
A trusted organization that issues public key certificates and provides identification to the bearer.

client-certificate authentication
An authentication mechanism that uses HTTP over SSL, in which the server and, optionally, the client authenticate each other with a public key certificate that conforms to a standard that is defined by X.509 Public Key Infrastructure.

comment
In an XML document, text that is ignored unless the parser is specifically told to recognize it.

commit
The point in a transaction when all updates to any resources involved in the transaction are made permanent.

component
See *J2EE component*.

component (JavaServer Faces technology)
See *JavaServer Faces UI component.*

component contract
The contract between a J2EE component and its container. The contract includes life-cycle management of the component, a context interface that the instance uses to obtain various information and services from its container, and a list of services that every container must provide for its components.

component-managed sign-on
A mechanism whereby security information needed for signing on to a resource is provided by an application component.

connection
See *resource manager connection*.

connection factory
See *resource manager connection factory.*

connector

A standard extension mechanism for containers that provides connectivity to enterprise information systems. A connector is specific to an enterprise information system and consists of a resource adapter and application development tools for enterprise information system connectivity. The resource adapter is plugged in to a container through its support for system-level contracts defined in the Connector architecture.

Connector architecture

An architecture for integration of J2EE products with enterprise information systems. There are two parts to this architecture: a resource adapter provided by an enterprise information system vendor and the J2EE product that allows this resource adapter to plug in. This architecture defines a set of contracts that a resource adapter must support to plug in to a J2EE product—for example, transactions, security, and resource management.

container

An entity that provides life-cycle management, security, deployment, and runtime services to J2EE components. Each type of container (EJB, web, JSP, servlet, applet, and application client) also provides component-specific services.

container-managed persistence

The mechanism whereby data transfer between an entity bean's variables and a resource manager is managed by the entity bean's container.

container-managed sign-on

The mechanism whereby security information needed for signing on to a resource is supplied by the container.

container-managed transaction

A transaction whose boundaries are defined by an EJB container. An entity bean must use container-managed transactions.

content

In an XML document, the part that occurs after the prolog, including the root element and everything it contains.

context attribute

An object bound into the context associated with a servlet.

context root

A name that gets mapped to the document root of a web application.

conversational state

The field values of a session bean plus the transitive closure of the objects reachable from the bean's fields. The transitive closure of a bean is defined in terms of the serialization protocol for the Java programming language, that is, the fields that would be stored by serializing the bean instance.

CORBA
Common Object Request Broker Architecture. A language-independent distributed object model specified by the OMG.

create **method**
A method defined in the home interface and invoked by a client to create an enterprise bean.

credentials
The information describing the security attributes of a principal.

CSS
Cascading style sheet. A stylesheet used with HTML and XML documents to add a style to all elements marked with a particular tag, for the direction of browsers or other presentation mechanisms.

CTS
Compatibility test suite. A suite of compatibility tests for verifying that a J2EE product complies with the J2EE platform specification.

data
The contents of an element in an XML stream, generally used when the element does not contain any subelements. When it does, the term *content* is generally used. When the only text in an XML structure is contained in simple elements and when elements that have subelements have little or no data mixed in, then that structure is often thought of as XML data, as opposed to an XML document.

DDP
Document-driven programming. The use of XML to define applications.

declaration
The very first thing in an XML document, which declares it as XML. The minimal declaration is <?xml version="1.0"?>. The declaration is part of the document prolog.

declarative security
Mechanisms used in an application that are expressed in a declarative syntax in a deployment descriptor.

delegation
An act whereby one principal authorizes another principal to use its identity or privileges with some restrictions.

deployer
A person who installs J2EE modules and applications into an operational environment.

deployment
The process whereby software is installed into an operational environment.

deployment descriptor
An XML file provided with each module and J2EE application that describes how they should be deployed. The deployment descriptor directs a deployment tool to deploy a module or application with specific container options and describes specific configuration requirements that a deployer must resolve.

destination
A JMS administered object that encapsulates the identity of a JMS queue or topic. See *point-to-point messaging system*, *publish/subscribe messaging system*.

digest authentication
An authentication mechanism in which a web application authenticates itself to a web server by sending the server a message digest along with its HTTP request message. The digest is computed by employing a one-way hash algorithm to a concatenation of the HTTP request message and the client's password. The digest is typically much smaller than the HTTP request and doesn't contain the password.

distributed application
An application made up of distinct components running in separate runtime environments, usually on different platforms connected via a network. Typical distributed applications are two-tier (client-server), three-tier (client-middleware-server), and multitier (client-multiple middleware-multiple servers).

document
In general, an XML structure in which one or more elements contains text intermixed with subelements. See also *data*.

Document Object Model
An API for accessing and manipulating XML documents as tree structures. DOM provides platform-neutral, language-neutral interfaces that enables programs and scripts to dynamically access and modify content and structure in XML documents.

document root
The top-level directory of a WAR. The document root is where JSP pages, client-side classes and archives, and static web resources are stored.

DOM
See *Document Object Model*.

DTD
Document type definition. An optional part of the XML document prolog, as specified by the XML standard. The DTD specifies constraints on the valid tags and tag sequences that can be in the document. The DTD has a number of shortcomings, however, and this has led to various schema proposals. For example, the DTD entry `<!ELEMENT username (#PCDATA)>` says that the XML element called `username` contains parsed character data—that is, text alone, with no other structural elements under it. The DTD includes both the local subset, defined in the current file, and the external subset, which consists of the definitions contained in external DTD files that are referenced in the local subset using a parameter entity.

durable subscription
In a JMS publish/subscribe messaging system, a subscription that continues to exist whether or not there is a current active subscriber object. If there is no active subscriber, the JMS provider retains the subscription's messages until they are received by the subscription or until they expire.

EAR file
Enterprise Archive file. A JAR archive that contains a J2EE application.

ebXML
Electronic Business XML. A group of specifications designed to enable enterprises to conduct business through the exchange of XML-based messages. It is sponsored by OASIS and the United Nations Centre for the Facilitation of Procedures and Practices in Administration, Commerce and Transport (U.N./CEFACT).

EJB
See *Enterprise JavaBeans*.

EJB container
A container that implements the EJB component contract of the J2EE architecture. This contract specifies a runtime environment for enterprise beans that includes security, concurrency, life-cycle management, transactions, deployment, naming, and other services. An EJB container is provided by an EJB or J2EE server.

EJB container provider
A vendor that supplies an EJB container.

EJB context
An object that allows an enterprise bean to invoke services provided by the container and to obtain the information about the caller of a client-invoked method.

EJB home object
An object that provides the life-cycle operations (create, remove, find) for an enterprise bean. The class for the EJB home object is generated by the container's deployment tools. The EJB home object implements the enterprise bean's home interface. The client references an EJB home object to perform life-cycle operations on an EJB object. The client uses JNDI to locate an EJB home object.

EJB JAR file
A JAR archive that contains an EJB module.

EJB module
A deployable unit that consists of one or more enterprise beans and an EJB deployment descriptor.

EJB object
An object whose class implements the enterprise bean's remote interface. A client never references an enterprise bean instance directly; a client always references an EJB object. The class of an EJB object is generated by a container's deployment tools.

EJB server
Software that provides services to an EJB container. For example, an EJB container typically relies on a transaction manager that is part of the EJB server to perform the two-phase commit across all the participating resource managers. The J2EE architecture assumes that an EJB container is hosted by an EJB server from the same vendor, so it does not specify the contract between these two entities. An EJB server can host one or more EJB containers.

EJB server provider
A vendor that supplies an EJB server.

element
A unit of XML data, delimited by tags. An XML element can enclose other elements.

empty tag
A tag that does not enclose any content.

enterprise bean
A J2EE component that implements a business task or business entity and is hosted by an EJB container; either an entity bean, a session bean, or a message-driven bean.

enterprise bean provider
An application developer who produces enterprise bean classes, remote and home interfaces, and deployment descriptor files, and packages them in an EJB JAR file.

enterprise information system
The applications that constitute an enterprise's existing system for handling companywide information. These applications provide an information infrastructure for an enterprise. An enterprise information system offers a well-defined set of services to its clients. These services are exposed to clients as local or remote interfaces or both. Examples of enterprise information systems include enterprise resource planning systems, mainframe transaction processing systems, and legacy database systems.

enterprise information system resource
An entity that provides enterprise information system-specific functionality to its clients. Examples are a record or set of records in a database system, a business object in an enterprise resource planning system, and a transaction program in a transaction processing system.

Enterprise JavaBeans (EJB)
A component architecture for the development and deployment of object-oriented, distributed, enterprise-level applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and secure.

Enterprise JavaBeans Query Language (EJB QL)
Defines the queries for the finder and select methods of an entity bean having container-managed persistence. A subset of SQL92, EJB QL has extensions that allow navigation over the relationships defined in an entity bean's abstract schema.

entity
A distinct, individual item that can be included in an XML document by referencing it. Such an entity reference can name an entity as small as a character (for example, `&lt;`, which references the less-than symbol or left angle bracket, <). An entity reference can also reference an entire document, an external entity, or a collection of DTD definitions.

entity bean
An enterprise bean that represents persistent data maintained in a database. An entity bean can manage its own persistence or can delegate this function to its container. An entity bean is identified by a primary key. If the container in which an entity bean is hosted crashes, the entity bean, its primary key, and any remote references survive the crash.

entity reference
A reference to an entity that is substituted for the reference when the XML document is parsed. It can reference a predefined entity such as &lt; or reference one that is defined in the DTD. In the XML data, the reference could be to an entity that is defined in the local subset of the DTD or to an external XML file (an external entity). The DTD can also carve out a segment of DTD specifications and give it a name so that it can be reused (included) at multiple points in the DTD by defining a parameter entity.

error
A SAX parsing error is generally a validation error; in other words, it occurs when an XML document is not valid, although it can also occur if the declaration specifies an XML version that the parser cannot handle. See also *fatal error*, *warning*.

Extensible Markup Language
See *XML*.

external entity
An entity that exists as an external XML file, which is included in the XML document using an entity reference.

external subset
That part of a DTD that is defined by references to external DTD files.

fatal error
A fatal error occurs in the SAX parser when a document is not well formed or otherwise cannot be processed. See also *error*, *warning*.

filter
An object that can transform the header or content (or both) of a request or response. Filters differ from web components in that they usually do not themselves create responses but rather modify or adapt the requests for a resource, and modify or adapt responses from a resource. A filter should not have any dependencies on a web resource for which it is acting as a filter so that it can be composable with more than one type of web resource.

filter chain
A concatenation of XSLT transformations in which the output of one transformation becomes the input of the next.

finder method
A method defined in the home interface and invoked by a client to locate an entity bean.

form-based authentication
An authentication mechanism in which a web container provides an application-specific form for logging in. This form of authentication uses Base64 encoding and can expose user names and passwords unless all connections are over SSL.

general entity
An entity that is referenced as part of an XML document's content, as distinct from a parameter entity, which is referenced in the DTD. A general entity can be a parsed entity or an unparsed entity.

group
An authenticated set of users classified by common traits such as job title or customer profile. Groups are also associated with a set of roles, and every user that is a member of a group inherits all the roles assigned to that group.

handle
An object that identifies an enterprise bean. A client can serialize the handle and then later deserialize it to obtain a reference to the enterprise bean.

home handle
An object that can be used to obtain a reference to the home interface. A home handle can be serialized and written to stable storage and deserialized to obtain the reference.

home interface
One of two interfaces for an enterprise bean. The home interface defines zero or more methods for managing an enterprise bean. The home interface of a session bean defines create and remove methods, whereas the home interface of an entity bean defines create, finder, and remove methods.

HTML
Hypertext Markup Language. A markup language for hypertext documents on the Internet. HTML enables the embedding of images, sounds, video streams, form fields, references to other objects with URLs, and basic text formatting.

HTTP
Hypertext Transfer Protocol. The Internet protocol used to retrieve hypertext objects from remote hosts. HTTP messages consist of requests from client to server and responses from server to client.

HTTPS
HTTP layered over the SSL protocol.

IDL

Interface Definition Language. A language used to define interfaces to remote CORBA objects. The interfaces are independent of operating systems and programming languages.

IIOP

Internet Inter-ORB Protocol. A protocol used for communication between CORBA object request brokers.

impersonation

An act whereby one entity assumes the identity and privileges of another entity without restrictions and without any indication visible to the recipients of the impersonator's calls that delegation has taken place. Impersonation is a case of simple delegation.

initialization parameter

A parameter that initializes the context associated with a servlet.

ISO 3166

The international standard for country codes maintained by the International Organization for Standardization (ISO).

ISV

Independent software vendor.

J2EE

See *Java 2 Platform, Enterprise Edition.*

J2EE application

Any deployable unit of J2EE functionality. This can be a single J2EE module or a group of modules packaged into an EAR file along with a J2EE application deployment descriptor. J2EE applications are typically engineered to be distributed across multiple computing tiers.

J2EE component

A self-contained functional software unit supported by a container and configurable at deployment time. The J2EE specification defines the following J2EE components:

- Application clients and applets are components that run on the client.
- Java servlet and JavaServer Pages (JSP) technology components are web components that run on the server.
- Enterprise JavaBeans (EJB) components (enterprise beans) are business components that run on the server.

J2EE components are written in the Java programming language and are compiled in the same way as any program in the language. The difference between J2EE components and "standard" Java classes is that J2EE components are assembled into a J2EE application, verified to be well formed and in compliance with the J2EE specification, and deployed to production, where they are run and managed by the J2EE server or client container.

J2EE module
A software unit that consists of one or more J2EE components of the same container type and one deployment descriptor of that type. There are four types of modules: EJB, web, application client, and resource adapter. Modules can be deployed as stand-alone units or can be assembled into a J2EE application.

J2EE product
An implementation that conforms to the J2EE platform specification.

J2EE product provider
A vendor that supplies a J2EE product.

J2EE server
The runtime portion of a J2EE product. A J2EE server provides EJB or web containers or both.

J2ME
See *Java 2 Platform, Micro Edition*.

J2SE
See *Java 2 Platform, Standard Edition*.

JAR
Java archive. A platform-independent file format that permits many files to be aggregated into one file.

Java 2 Platform, Enterprise Edition (J2EE)
An environment for developing and deploying enterprise applications. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing multitiered, web-based applications.

Java 2 Platform, Micro Edition (J2ME)
A highly optimized Java runtime environment targeting a wide range of consumer products, including pagers, cellular phones, screen phones, digital set-top boxes, and car navigation systems.

Java 2 Platform, Standard Edition (J2SE)
The core Java technology platform.

Java API for XML Processing (JAXP)
An API for processing XML documents. JAXP leverages the parser standards SAX and DOM so that you can choose to parse your data as a stream of events or to build a tree-structured representation of it. JAXP supports the

XSLT standard, giving you control over the presentation of the data and enabling you to convert the data to other XML documents or to other formats, such as HTML. JAXP provides namespace support, allowing you to work with schema that might otherwise have naming conflicts.

Java API for XML Registries (JAXR)
An API for accessing various kinds of XML registries.

Java API for XML-based RPC (JAX-RPC)
An API for building web services and clients that use remote procedure calls and XML.

Java IDL
A technology that provides CORBA interoperability and connectivity capabilities for the J2EE platform. These capabilities enable J2EE applications to invoke operations on remote network services using the Object Management Group IDL and IIOP.

Java Message Service (JMS)
An API for invoking operations on enterprise messaging systems.

Java Naming and Directory Interface (JNDI)
An API that provides naming and directory functionality.

Java Secure Socket Extension (JSSE)
A set of packages that enable secure Internet communications.

Java Transaction API (JTA)
An API that allows applications and J2EE servers to access transactions.

Java Transaction Service (JTS)
Specifies the implementation of a transaction manager that supports JTA and implements the Java mapping of the Object Management Group Object Transaction Service 1.1 specification at the level below the API.

JavaBeans component
A Java class that can be manipulated by tools and composed into applications. A JavaBeans component must adhere to certain property and event interface conventions.

JavaMail
An API for sending and receiving email.

JavaServer Faces
A framework for building server-side user interfaces for web applications written in the Java programming language.

JavaServer Faces conversion model
A mechanism for converting between string-based markup generated by JavaServer Faces UI components and server-side Java objects.

JavaServer Faces event and listener model
A mechanism for determining how events emitted by JavaServer Faces UI components are handled. This model is based on the JavaBeans component event and listener model.

JavaServer Faces expression language
A simple expression language used by a JavaServer Faces UI component tag attributes to bind the associated component to a bean property or to bind the associated component's value to a method or an external data source, such as a bean property. Unlike JSP EL expressions, JavaServer Faces EL expressions are evaluated by the JavaServer Faces implementation rather than by the web container.

JavaServer Faces navigation model
A mechanism for defining the sequence in which pages in a JavaServer Faces application are displayed.

JavaServer Faces UI component
A user interface control that outputs data to a client or allows a user to input data to a JavaServer Faces application.

JavaServer Faces UI component class
A JavaServer Faces class that defines the behavior and properties of a JavaServer Faces UI component.

JavaServer Faces validation model
A mechanism for validating the data a user inputs to a JavaServer Faces UI component.

JavaServer Pages (JSP)
An extensible web technology that uses static data, JSP elements, and server-side Java objects to generate dynamic content for a client. Typically the static data is HTML or XML elements, and in many cases the client is a web browser.

JavaServer Pages Standard Tag Library (JSTL)
A tag library that encapsulates core functionality common to many JSP applications. JSTL has support for common, structural tasks such as iteration and conditionals, tags for manipulating XML documents, internationalization and locale-specific formatting tags, SQL tags, and functions.

JAXR client
A client program that uses the JAXR API to access a business registry via a JAXR provider.

JAXR provider
An implementation of the JAXR API that provides access to a specific registry provider or to a class of registry providers that are based on a common specification.

JDBC
An API for database-independent connectivity between the J2EE platform and a wide range of data sources.

JMS
See *Java Message Service*.

JMS administered object
A preconfigured JMS object (a resource manager connection factory or a destination) created by an administrator for the use of JMS clients and placed in a JNDI namespace.

JMS application
One or more JMS clients that exchange messages.

JMS client
A Java language program that sends or receives messages.

JMS provider
A messaging system that implements the Java Message Service as well as other administrative and control functionality needed in a full-featured messaging product.

JMS session
A single-threaded context for sending and receiving JMS messages. A JMS session can be nontransacted, locally transacted, or participating in a distributed transaction.

JNDI
See *Java Naming and Directory Interface*.

JSP
See *JavaServer Pages*.

JSP action
A JSP element that can act on implicit objects and other server-side objects or can define new scripting variables. Actions follow the XML syntax for elements, with a start tag, a body, and an end tag; if the body is empty it can also use the empty tag syntax. The tag must use a prefix. There are standard and custom actions.

JSP container
A container that provides the same services as a servlet container and an engine that interprets and processes JSP pages into a servlet.

JSP container, distributed
A JSP container that can run a web application that is tagged as distributable and is spread across multiple Java virtual machines that might be running on different hosts.

JSP custom action
A user-defined action described in a portable manner by a tag library descriptor and imported into a JSP page by a taglib directive. Custom actions are used to encapsulate recurring tasks in writing JSP pages.

JSP custom tag
A tag that references a JSP custom action.

JSP declaration
A JSP scripting element that declares methods, variables, or both in a JSP page.

JSP directive
A JSP element that gives an instruction to the JSP container and is interpreted at translation time.

JSP document
A JSP page written in XML syntax and subject to the constraints of XML documents.

JSP element
A portion of a JSP page that is recognized by a JSP translator. An element can be a directive, an action, or a scripting element.

JSP expression
A scripting element that contains a valid scripting language expression that is evaluated, converted to a String, and placed into the implicit out object.

JSP expression language
A language used to write expressions that access the properties of JavaBeans components. EL expressions can be used in static text and in any standard or custom tag attribute that can accept an expression.

JSP page
A text-based document containing static text and JSP elements that describes how to process a request to create a response. A JSP page is translated into and handles requests as a servlet.

JSP scripting element
A JSP declaration, scriptlet, or expression whose syntax is defined by the JSP specification and whose content is written according to the scripting language used in the JSP page. The JSP specification describes the syntax and semantics for the case where the language page attribute is "java".

JSP scriptlet

A JSP scripting element containing any code fragment that is valid in the scripting language used in the JSP page. The JSP specification describes what is a valid scriptlet for the case where the language page attribute is "java".

JSP standard action

An action that is defined in the JSP specification and is always available to a JSP page.

JSP tag file

A source file containing a reusable fragment of JSP code that is translated into a tag handler when a JSP page is translated into a servlet.

JSP tag handler

A Java programming language object that implements the behavior of a custom tag.

JSP tag library

A collection of custom tags described via a tag library descriptor and Java classes.

JSTL

See *JavaServer Pages Standard Tag Library.*

JTA

See *Java Transaction API*.

JTS

See *Java Transaction Service*.

keystore

A file containing the keys and certificates used for authentication.

life cycle (J2EE component)

The framework events of a J2EE component's existence. Each type of component has defining events that mark its transition into states in which it has varying availability for use. For example, a servlet is created and has its init method called by its container before invocation of its service method by clients or other servlets that require its functionality. After the call of its init method, it has the data and readiness for its intended use. The servlet's destroy method is called by its container before the ending of its existence so that processing associated with winding up can be done and resources can be released. The init and destroy methods in this example are callback methods. Similar considerations apply to the life cycle of all J2EE component types: enterprise beans, web components (servlets or JSP pages), applets, and application clients.

life cycle (JavaServer Faces)

A set of phases during which a request for a page is received, a UI component tree representing the page is processed, and a response is produced. During the phases of the life cycle:

- The local data of the components is updated with the values contained in the request parameters.
- Events generated by the components are processed.
- Validators and converters registered on the components are processed.
- The components' local data is updated to back-end objects.
- The response is rendered to the client while the component state of the response is saved on the server for future requests.

local subset

That part of the DTD that is defined within the current XML file.

managed bean creation facility

A mechanism for defining the characteristics of JavaBeans components used in a JavaServer Faces application.

message

In the Java Message Service, an asynchronous request, report, or event that is created, sent, and consumed by an enterprise application and not by a human. It contains vital information needed to coordinate enterprise applications, in the form of precisely formatted data that describes specific business actions.

message consumer

An object created by a JMS session that is used for receiving messages sent to a destination.

message-driven bean

An enterprise bean that is an asynchronous message consumer. A message-driven bean has no state for a specific client, but its instance variables can contain state across the handling of client messages, including an open database connection and an object reference to an EJB object. A client accesses a message-driven bean by sending messages to the destination for which the bean is a message listener.

message producer

An object created by a JMS session that is used for sending messages to a destination.

mixed-content model

A DTD specification that defines an element as containing a mixture of text and one more other elements. The specification must start with #PCDATA, followed by diverse elements, and must end with the “zero-or-more” asterisk symbol (*).

method-binding expression

A JavaServer Faces EL expression that refers to a method of a backing bean. This method performs either event handling, validation, or navigation processing for the UI component whose tag uses the method-binding expression.

method permission

An authorization rule that determines who is permitted to execute one or more enterprise bean methods.

mutual authentication

An authentication mechanism employed by two parties for the purpose of proving each other’s identity to one another.

namespace

A standard that lets you specify a unique label for the set of element names defined by a DTD. A document using that DTD can be included in any other document without having a conflict between element names. The elements defined in your DTD are then uniquely identified so that, for example, the parser can tell when an element <name> should be interpreted according to your DTD rather than using the definition for an element <name> in a different DTD.

naming context

A set of associations between unique, atomic, people-friendly identifiers and objects.

naming environment

A mechanism that allows a component to be customized without the need to access or change the component’s source code. A container implements the component’s naming environment and provides it to the component as a JNDI naming context. Each component names and accesses its environment entries using the java:comp/env JNDI context. The environment entries are declaratively specified in the component’s deployment descriptor.

normalization

The process of removing redundancy by modularizing, as with subroutines, and of removing superfluous differences by reducing them to a common denominator. For example, line endings from different systems are normalized by reducing them to a single new line, and multiple whitespace characters are normalized to one space.

North American Industry Classification System (NAICS)
A system for classifying business establishments based on the processes they use to produce goods or services.

notation
A mechanism for defining a data format for a non-XML document referenced as an unparsed entity. This is a holdover from SGML. A newer standard is to use MIME data types and namespaces to prevent naming conflicts.

OASIS
Organization for the Advancement of Structured Information Standards. A consortium that drives the development, convergence, and adoption of e-business standards. Its web site is http://www.oasis-open.org/. The DTD repository it sponsors is at http://www.XML.org.

OMG
Object Management Group. A consortium that produces and maintains computer industry specifications for interoperable enterprise applications. Its web site is http://www.omg.org/.

one-way messaging
A method of transmitting messages without having to block until a response is received.

ORB
Object request broker. A library that enables CORBA objects to locate and communicate with one another.

OS principal
A principal native to the operating system on which the J2EE platform is executing.

OTS
Object Transaction Service. A definition of the interfaces that permit CORBA objects to participate in transactions.

parameter entity
An entity that consists of DTD specifications, as distinct from a general entity. A parameter entity defined in the DTD can then be referenced at other points, thereby eliminating the need to recode the definition at each location it is used.

parsed entity
A general entity that contains XML and therefore is parsed when inserted into the XML document, as opposed to an unparsed entity.

parser

A module that reads in XML data from an input source and breaks it into chunks so that your program knows when it is working with a tag, an attribute, or element data. A nonvalidating parser ensures that the XML data is well formed but does not verify that it is valid. See also *validating parser.*

passivation

The process of transferring an enterprise bean from memory to secondary storage. See *activation*.

persistence

The protocol for transferring the state of an entity bean between its instance variables and an underlying database.

persistent field

A virtual field of an entity bean that has container-managed persistence; it is stored in a database.

POA

Portable Object Adapter. A CORBA standard for building server-side applications that are portable across heterogeneous ORBs.

point-to-point messaging system

A messaging system built on the concept of message queues. Each message is addressed to a specific queue; clients extract messages from the queues established to hold their messages.

primary key

An object that uniquely identifies an entity bean within a home.

principal

The identity assigned to a user as a result of authentication.

privilege

A security attribute that does not have the property of uniqueness and that can be shared by many principals.

processing instruction

Information contained in an XML structure that is intended to be interpreted by a specific application.

programmatic security

Security decisions that are made by security-aware applications. Programmatic security is useful when declarative security alone is not sufficient to express the security model of an application.

prolog

The part of an XML document that precedes the XML data. The prolog includes the declaration and an optional DTD.

public key certificate

Used in client-certificate authentication to enable the server, and optionally the client, to authenticate each other. The public key certificate is the digital equivalent of a passport. It is issued by a trusted organization, called a certificate authority, and provides identification for the bearer.

publish/subscribe messaging system

A messaging system in which clients address messages to a specific node in a content hierarchy, called a topic. Publishers and subscribers are generally anonymous and can dynamically publish or subscribe to the content hierarchy. The system takes care of distributing the messages arriving from a node's multiple publishers to its multiple subscribers.

query string

A component of an HTTP request URL that contains a set of parameters and values that affect the handling of the request.

queue

See *point-to-point messaging system*.

RAR

Resource Adapter Archive. A JAR archive that contains a resource adapter module.

RDF

Resource Description Framework. A standard for defining the kind of data that an XML file contains. Such information can help ensure semantic integrity—for example—by helping to make sure that a date is treated as a date rather than simply as text.

RDF schema

A standard for specifying consistency rules that apply to the specifications contained in an RDF.

realm

See *security policy domain*. Also, a string, passed as part of an HTTP request during basic authentication, that defines a protection space. The protected resources on a server can be partitioned into a set of protection spaces, each with its own authentication scheme or authorization database or both.

In the J2EE server authentication service, a realm is a complete database of roles, users, and groups that identify valid users of a web application or a set of web applications.

reentrant entity bean

An entity bean that can handle multiple simultaneous, interleaved, or nested invocations that will not interfere with each other.

reference
See *entity reference*.

registry
An infrastructure that enables the building, deployment, and discovery of web services. It is a neutral third party that facilitates dynamic and loosely coupled business-to-business (B2B) interactions.

registry provider
An implementation of a business registry that conforms to a specification for XML registries (for example, ebXML or UDDI).

relationship field
A virtual field of an entity bean having container-managed persistence; it identifies a related entity bean.

remote interface
One of two interfaces for an enterprise bean. The remote interface defines the business methods callable by a client.

remove **method**
Method defined in the home interface and invoked by a client to destroy an enterprise bean.

render kit
A set of renderers that render output to a particular client. The JavaServer Faces implementation provides a standard HTML render kit, which is composed of renderers that can render HMTL markup.

renderer
A Java class that can render the output for a set of JavaServer Faces UI components.

request-response messaging
A method of messaging that includes blocking until a response is received.

resource adapter
A system-level software driver that is used by an EJB container or an application client to connect to an enterprise information system. A resource adapter typically is specific to an enterprise information system. It is available as a library and is used within the address space of the server or client using it. A resource adapter plugs in to a container. The application components deployed on the container then use the client API (exposed by the adapter) or tool-generated high-level abstractions to access the underlying enterprise information system. The resource adapter and EJB container collaborate to provide the underlying mechanisms—transactions, security, and connection pooling—for connectivity to the enterprise information system.

resource adapter module
A deployable unit that contains all Java interfaces, classes, and native libraries, implementing a resource adapter along with the resource adapter deployment descriptor.

resource manager
Provides access to a set of shared resources. A resource manager participates in transactions that are externally controlled and coordinated by a transaction manager. A resource manager typically is in a different address space or on a different machine from the clients that access it. Note: An enterprise information system is referred to as a resource manager when it is mentioned in the context of resource and transaction management.

resource manager connection
An object that represents a session with a resource manager.

resource manager connection factory
An object used for creating a resource manager connection.

RMI
Remote Method Invocation. A technology that allows an object running in one Java virtual machine to invoke methods on an object running in a different Java virtual machine.

RMI-IIOP
A version of RMI implemented to use the CORBA IIOP protocol. RMI over IIOP provides interoperability with CORBA objects implemented in any language if all the remote interfaces are originally defined as RMI interfaces.

role (development)
The function performed by a party in the development and deployment phases of an application developed using J2EE technology. The roles are application component provider, application assembler, deployer, J2EE product provider, EJB container provider, EJB server provider, web container provider, web server provider, tool provider, and system administrator.

role mapping
The process of associating the groups or principals (or both), recognized by the container with security roles specified in the deployment descriptor. Security roles must be mapped by the deployer before a component is installed in the server.

role (security)
An abstract logical grouping of users that is defined by the application assembler. When an application is deployed, the roles are mapped to security identities, such as principals or groups, in the operational environment.

In the J2EE server authentication service, a role is an abstract name for permission to access a particular set of resources. A role can be compared to a key that can open a lock. Many people might have a copy of the key; the lock doesn't care who you are, only that you have the right key.

rollback
The point in a transaction when all updates to any resources involved in the transaction are reversed.

root
The outermost element in an XML document. The element that contains all other elements.

SAX
See *Simple API for XML*.

Simple API for XML
An event-driven interface in which the parser invokes one of several methods supplied by the caller when a parsing event occurs. Events include recognizing an XML tag, finding an error, encountering a reference to an external entity, or processing a DTD specification.

schema
A database-inspired method for specifying constraints on XML documents using an XML-based language. Schemas address deficiencies in DTDs, such as the inability to put constraints on the kinds of data that can occur in a particular field. Because schemas are founded on XML, they are hierarchical. Thus it is easier to create an unambiguous specification, and it is possible to determine the scope over which a comment is meant to apply.

Secure Socket Layer (SSL)
A technology that allows web browsers and web servers to communicate over a secured connection.

security attributes
A set of properties associated with a principal. Security attributes can be associated with a principal by an authentication protocol or by a J2EE product provider or both.

security constraint
A declarative way to annotate the intended protection of web content. A security constraint consists of a web resource collection, an authorization constraint, and a user data constraint.

security context
An object that encapsulates the shared state information regarding security between two entities.

security permission

A mechanism defined by J2SE, and used by the J2EE platform to express the programming restrictions imposed on application component developers.

security permission set

The minimum set of security permissions that a J2EE product provider must provide for the execution of each component type.

security policy domain

A scope over which security policies are defined and enforced by a security administrator. A security policy domain has a collection of users (or principals), uses a well-defined authentication protocol or protocols for authenticating users (or principals), and may have groups to simplify setting of security policies.

security role

See *role (security).*

security technology domain

A scope over which the same security mechanism is used to enforce a security policy. Multiple security policy domains can exist within a single technology domain.

security view

The set of security roles defined by the application assembler.

server certificate

Used with the HTTPS protocol to authenticate web applications. The certificate can be self-signed or approved by a certificate authority (CA). The HTTPS service of the Sun Java System Application Server Platform Edition 8.1 will not run unless a server certificate has been installed.

server principal

The OS principal that the server is executing as.

service element

A representation of the combination of one or more Connector components that share a single engine component for processing incoming requests.

service endpoint interface

A Java interface that declares the methods that a client can invoke on a web service.

servlet

A Java program that extends the functionality of a web server, generating dynamic content and interacting with web applications using a request-response paradigm.

servlet container

A container that provides the network services over which requests and responses are sent, decodes requests, and formats responses. All servlet containers must support HTTP as a protocol for requests and responses but can also support additional request-response protocols, such as HTTPS.

servlet container, distributed

A servlet container that can run a web application that is tagged as distributable and that executes across multiple Java virtual machines running on the same host or on different hosts.

servlet context

An object that contains a servlet's view of the web application within which the servlet is running. Using the context, a servlet can log events, obtain URL references to resources, and set and store attributes that other servlets in the context can use.

servlet mapping

Defines an association between a URL pattern and a servlet. The mapping is used to map requests to servlets.

session

An object used by a servlet to track a user's interaction with a web application across multiple HTTP requests.

session bean

An enterprise bean that is created by a client and that usually exists only for the duration of a single client-server session. A session bean performs operations, such as calculations or database access, for the client. Although a session bean can be transactional, it is not recoverable should a system crash occur. Session bean objects either can be stateless or can maintain conversational state across methods and transactions. If a session bean maintains state, then the EJB container manages this state if the object must be removed from memory. However, the session bean object itself must manage its own persistent data.

SGML

Standard Generalized Markup Language. The parent of both HTML and XML. Although HTML shares SGML's propensity for embedding presentation information in the markup, XML is a standard that allows information content to be totally separated from the mechanisms for rendering that content.

SOAP

Simple Object Access Protocol. A lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It defines, using XML technologies, an extensible messaging framework containing a message construct that can be exchanged over a variety of underlying protocols.

SOAP with Attachments API for Java (SAAJ)
The basic package for SOAP messaging, SAAJ contains the API for creating and populating a SOAP message.

SQL
Structured Query Language. The standardized relational database language for defining database objects and manipulating data.

SQL/J
A set of standards that includes specifications for embedding SQL statements in methods in the Java programming language and specifications for calling Java static methods as SQL stored procedures and user-defined functions. An SQL checker can detect errors in static SQL statements at program development time, rather than at execution time as with a JDBC driver.

SSL
Secure Socket Layer. A security protocol that provides privacy over the Internet. The protocol allows client-server applications to communicate in a way that cannot be eavesdropped upon or tampered with. Servers are always authenticated, and clients are optionally authenticated.

stateful session bean
A session bean with a conversational state.

stateless session bean
A session bean with no conversational state. All instances of a stateless session bean are identical.

system administrator
The person responsible for configuring and administering the enterprise's computers, networks, and software systems.

tag
In XML documents, a piece of text that describes a unit of data or an element. The tag is distinguishable as markup, as opposed to data, because it is surrounded by angle brackets (< and >). To treat such markup syntax as data, you use an entity reference or a CDATA section.

template
A set of formatting instructions that apply to the nodes selected by an XPath expression.

tool provider
An organization or software vendor that provides tools used for the development, packaging, and deployment of J2EE applications.

topic
See *publish-subscribe messaging system*.

transaction
An atomic unit of work that modifies data. A transaction encloses one or more program statements, all of which either complete or roll back. Transactions enable multiple users to access the same data concurrently.

transaction attribute
A value specified in an enterprise bean's deployment descriptor that is used by the EJB container to control the transaction scope when the enterprise bean's methods are invoked. A transaction attribute can have the following values: Required, RequiresNew, Supports, NotSupported, Mandatory, or Never.

transaction isolation level
The degree to which the intermediate state of the data being modified by a transaction is visible to other concurrent transactions and data being modified by other transactions is visible to it.

transaction manager
Provides the services and management functions required to support transaction demarcation, transactional resource management, synchronization, and transaction context propagation.

Unicode
A standard defined by the Unicode Consortium that uses a 16-bit code page that maps digits to characters in languages around the world. Because 16 bits covers 32,768 codes, Unicode is large enough to include all the world's languages, with the exception of ideographic languages that have a different character for every concept, such as Chinese. For more information, see http://www.unicode.org/.

Universal Description, Discovery and Integration (UDDI) project
An industry initiative to create a platform-independent, open framework for describing services, discovering businesses, and integrating business services using the Internet, as well as a registry. It is being developed by a vendor consortium.

Universal Standard Products and Services Classification (UNSPSC)
A schema that classifies and identifies commodities. It is used in sell-side and buy-side catalogs and as a standardized account code in analyzing expenditure.

unparsed entity

A general entity that contains something other than XML. By its nature, an unparsed entity contains binary data.

URI

Uniform resource identifier. A globally unique identifier for an abstract or physical resource. A URL is a kind of URI that specifies the retrieval protocol (http or https for web applications) and physical location of a resource (host name and host-relative path). A URN is another type of URI.

URL

Uniform resource locator. A standard for writing a textual reference to an arbitrary piece of data in the World Wide Web. A URL looks like this: protocol://host/localinfo where protocol specifies a protocol for fetching the object (such as http or ftp), host specifies the Internet name of the targeted host, and localinfo is a string (often a file name) passed to the protocol handler on the remote host.

URL path

The part of a URL passed by an HTTP request to invoke a servlet. A URL path consists of the context path + servlet path + path info, where

- Context path is the path prefix associated with a servlet context of which the servlet is a part. If this context is the default context rooted at the base of the web server's URL namespace, the path prefix will be an empty string. Otherwise, the path prefix starts with a / character but does not end with a / character.
- Servlet path is the path section that directly corresponds to the mapping that activated this request. This path starts with a / character.
- Path info is the part of the request path that is not part of the context path or the servlet path.

URN

Uniform resource name. A unique identifier that identifies an entity but doesn't tell where it is located. A system can use a URN to look up an entity locally before trying to find it on the web. It also allows the web location to change, while still allowing the entity to be found.

user data constraint

Indicates how data between a client and a web container should be protected. The protection can be the prevention of tampering with the data or prevention of eavesdropping on the data.

user (security)
An individual (or application program) identity that has been authenticated. A user can have a set of roles associated with that identity, which entitles the user to access all resources protected by those roles.

valid
A valid XML document, in addition to being well formed, conforms to all the constraints imposed by a DTD. It does not contain any tags that are not permitted by the DTD, and the order of the tags conforms to the DTD's specifications.

validating parser
A parser that ensures that an XML document is valid in addition to being well formed. See also *parser.*

value-binding expression
A JavaServer Faces EL expression that refers to a property of a backing bean. A component tag uses this expression to bind the associated component's value or the component instance to the bean property. If the component tag refers to the property via its value attribute, then the component's value is bound to the property. If the component tag refers to the property via its binding attribute then the component itself is bound to the property.

virtual host
Multiple hosts plus domain names mapped to a single IP address.

W3C
World Wide Web Consortium. The international body that governs Internet standards. Its web site is http://www.w3.org/.

WAR file
Web application archive file. A JAR archive that contains a web module.

warning
A SAX parser warning is generated when the document's DTD contains duplicate definitions and in similar situations that are not necessarily an error but which the document author might like to know about, because they could be. See also *fatal error, error.*

Web application
An application written for the Internet, including those built with Java technologies such as JavaServer Pages and servlets, as well as those built with non-Java technologies such as CGI and Perl.

Web application, distributable
A web application that uses J2EE technology written so that it can be deployed in a web container distributed across multiple Java virtual

machines running on the same host or different hosts. The deployment descriptor for such an application uses the distributable element.

Web component

A component that provides services in response to requests; either a servlet or a JSP page.

Web container

A container that implements the web component contract of the J2EE architecture. This contract specifies a runtime environment for web components that includes security, concurrency, life-cycle management, transaction, deployment, and other services. A web container provides the same services as a JSP container as well as a federated view of the J2EE platform APIs. A web container is provided by a web or J2EE server.

Web container, distributed

A web container that can run a web application that is tagged as distributable and that executes across multiple Java virtual machines running on the same host or on different hosts.

Web container provider

A vendor that supplies a web container.

Web module

A deployable unit that consists of one or more web components, other resources, and a web application deployment descriptor contained in a hierarchy of directories and files in a standard web application format.

Web resource

A static or dynamic object contained in a web application that can be referenced by a URL.

Web resource collection

A list of URL patterns and HTTP methods that describe a set of web resources to be protected.

Web server

Software that provides services to access the Internet, an intranet, or an extranet. A web server hosts web sites, provides support for HTTP and other protocols, and executes server-side programs (such as CGI scripts or servlets) that perform certain functions. In the J2EE architecture, a web server provides services to a web container. For example, a web container typically relies on a web server to provide HTTP message handling. The J2EE architecture assumes that a web container is hosted by a web server from the same vendor, so it does not specify the contract between these two entities. A web server can host one or more web containers.

Web server provider
A vendor that supplies a web server.

Web service
An application that exists in a distributed environment, such as the Internet. A web service accepts a request, performs its function based on the request, and returns a response. The request and the response can be part of the same operation, or they can occur separately, in which case the consumer does not need to wait for a response. Both the request and the response usually take the form of XML, a portable data-interchange format, and are delivered over a wire protocol, such as HTTP.

well-formed
An XML document that is syntactically correct. It does not have any angle brackets that are not part of tags, all tags have an ending tag or are themselves self-ending, and all tags are fully nested. Knowing that a document is well formed makes it possible to process it. However, a well-formed document may not be valid. To determine that, you need a validating parser and a DTD.

Xalan
An interpreting version of XSLT.

XHTML
An XML look-alike for HTML defined by one of several XHTML DTDs. To use XHTML for everything would of course defeat the purpose of XML, because the idea of XML is to identify information content, and not just to tell how to display it. You can reference it in a DTD, which allows you to say, for example, that the text in an element can contain <em> and <b> tags rather than being limited to plain text.

XLink
The part of the XLL specification that is concerned with specifying links between documents.

XLL
The XML Link Language specification, consisting of XLink and XPointer.

XML
Extensible Markup Language. A markup language that allows you to define the tags (markup) needed to identify the content, data, and text in XML documents. It differs from HTML, the markup language most often used to present information on the Internet. HTML has fixed tags that deal mainly with style or presentation. An XML document must undergo a transformation into a language with style tags under the control of a style sheet before it can be presented by a browser or other presentation mechanism. Two types

of style sheets used with XML are CSS and XSL. Typically, XML is transformed into HTML for presentation. Although tags can be defined as needed in the generation of an XML document, a document type definition (DTD) can be used to define the elements allowed in a particular type of document. A document can be compared by using the rules in the DTD to determine its validity and to locate particular elements in the document. A web services application's J2EE deployment descriptors are expressed in XML with schemas defining allowed elements. Programs for processing XML documents use SAX or DOM APIs.

XML registry
See *registry.*

XML Schema
The W3C specification for defining the structure, content, and semantics of XML documents.

XPath
An addressing mechanism for identifying the parts of an XML document.

XPointer
The part of the XLL specification that is concerned with identifying sections of documents so that they can be referenced in links or included in other documents.

XSL
Extensible Stylesheet Language. A standard that lets you do the following:

- Specify an addressing mechanism, so that you can identify the parts of an XML document that a transformation applies to (XPath).
- Specify tag conversions, so that you can convert XML data into different formats (XSLT).
- Specify display characteristics, such page sizes, margins, and font heights and widths, as well as the flow objects on each page. Information fills in one area of a page and then automatically flows to the next object when that area fills up. That allows you to wrap text around pictures, for example, or to continue a newsletter article on a different page (XSL-FO).

XSL-FO
A subcomponent of XSL used for describing font sizes, page layouts, and how information flows from one page to another.

XSLT
Extensible Stylesheet Language Transformations. An XML document that controls the transformation of an XML document into another XML document or HTML. The target document often has presentation-related tags dic-

tating how it will be rendered by a browser or other presentation mechanism. XSLT was formerly a part of XSL, which also included a tag language of style flow objects.

XSLTC
A compiling version of XSLT.

Index